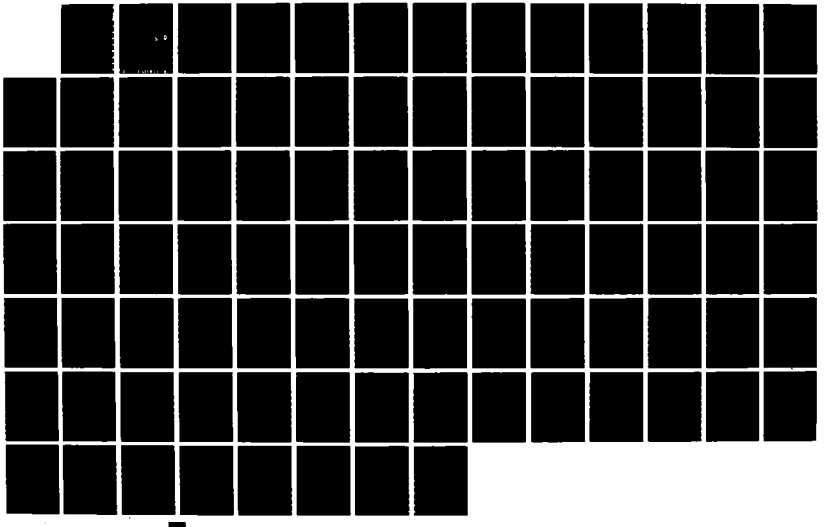
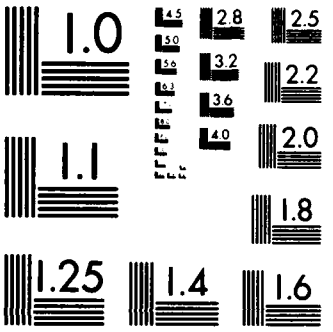
AD-A170 741 AUTOMATED GENERATION OF INPUT OUTPUT PAIRS FOR THE CAIS 1/1
VALIDATION TEST SUITE(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH J R JENKINS MAY 86
UNCLASSIFIED AFIT/CI/NR-86-76T F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 86-76T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Automated Generation of Input Output Pairs For The Cais Validation Test Suite		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joyce Rene Jenkins		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Arizona State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1986 13. NUMBER OF PAGES 78
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLAS 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE AUG 13 1986 S D E

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 6 AUG 86
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

ATTACHED.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

86 8 1 017

AD-A170 741

DTIC FILE COPY

ABSTRACT

A method designed by Lindquist and Facemire [1] to construct validation tests for the Common APSE Interface Set (CAIS), a set of kernel interfaces which allow for the transportability of the Ada Programming Support Environment (APSE) tools, utilizes White-box testing methodologies in a Black-box fashion. A White-box testing approach is taken to examine the internal structure of the Ada-based Abstract Machine specification of the CAIS. Because the implementation details of the CAIS are not known at validation time and the same validation suite is to be used on several CAIS implementations, Black-box testing is a more desirable approach. Their solution is to identify test cases by analyzing the program using a White-box approach. Tests cases can then be constructed and applied in a Black-box fashion. The system described in this paper uses symbolic execution to create an execution tree which identifies all execution paths through the program. The path condition, a label corresponding to each execution path, identifies the conditions which cause that path to be executed. For each path, symbolic execution produces a range over the input interface parameters and a list of corresponding altered outputs [1]. This range and list of altered outputs are known as Input/Output(I/O) pairs. The I/O pairs are then used to construct the tests needed for validating the CAIS. This paper discusses the detailed design and preliminary implementation of IOGEN, a system for generating the I/O pairs for the CAIS using symbolic execution.

AUTOMATED GENERATION OF INPUT/OUTPUT PAIRS FOR THE CAIS VALIDATION TEST SUITE

by

Joyce Rene' Jenkins

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 1986

AUTOMATED GENERATION OF INPUT/OUTPUT PAIRS

FOR THE CAIS VALIDATION TEST SUITE

by

Joyce Rene' Jenkins

has been approved

April 1986

APPROVED:

Timothy E Lindquist ,Chairperson

James S. Collofello

Terry F. Mellon

Supervisory Committee

ACCEPTED:

Department Chairperson

Dean, Graduate College

Abstract

This thesis addresses automating the process of generating test cases. The detailed design and preliminary implementation of a system for generating the input/output pairs are presented. The system, IOGEN, will be used for constructing validation tests for the Common APSE Interface Set (CAIS). The input/output pairs generated using symbolic execution and an overview of the symbolic execution technique as it applies to testing are given. Finally, possible areas for enhancing the IOGEN system are provided.

Dedication

To my mother and father.

Acknowledgement

I would like to thank Dr. Timothy Lindquist for his invaluable guidance and patience throughout the development of this thesis. My thanks to Dr. James Collofello and Dr. Terry Mellon for their participation as committee members and Marc Lesure for his assistance with RRIPLL. Finally, I would like to thank Ronald Harden for all the moral support he provided.

Table of Contents

List of Figures

FIGURE

FIGURE (continued)

CHAPTER 1

Introduction

The Ada* Program, developed at the initiative of the United States Department of Defense (DoD), adopted the concept that a common environment which supports automated tools is essential to the computer growth within the military service [4]. From this concept evolved the development of a Programming Support Environment (PSE). Within this environment resides software tools such as compilers, editors, debuggers, configuration control aids, linkers, and text formatters. All of these tools are designed to aid programming tasks during the coding, testing, and debugging phase.

The Ada Programming Support Environment (APSE), implemented in Ada, was designed to support mission critical software written in Ada. It was the intent of the Department of Defense that the APSEs become the basic life-cycle environment for all mission critical computer systems (MCCS) [1]. APSEs were therefore expected to be transported among various types of machines and operating systems. Transporting tools among various APSE architectures realized one of the fundamental goals of the Ada program - to increase the transportability and maintainability of embedded software systems [5]. However, the Ada language does not provide a means for communications between APSE

*Ada is a registered trademark of the United States Government (Ada Joint Program Office)

tools and the host system. A set of kernel interfaces were therefore necessary.

To formulate the requirements for the kernel interfaces for tools needed to support the APSE, the Ada Joint Program Office (AJPO) formed the Kernel APSE Interface Team (KIT), chaired by the Naval Ocean Systems Center, and the Kernel APSE Interface Team from Industry and Academia (KITIA). The KIT/KITIA have designed a set of interfaces which allow APSE tools to access the facilities and services of the host system. This set of interfaces is called the Common APSE Interface Set (CAIS). The CAIS is not designed to encompass total operating system's functionality, but instead it includes facilities which are most useful to tools [7].

In early 1986, theDepartment of Defense proposed a draft government standard CAIS. The purpose for standardization is to promote transportability of Ada tools across all DoD-sanctioned APSE's. As a result, the APSE Evaluation and Validation Team (E&V) is responsible for initiating the development of a CAIS Validation Capability (CVC) [5]. The CVC will be designed to test whether the implementation of the CAIS adheres strictly to the specifications.

Facemire and Lindquist [5] described a specification and validation technique to construct validation tests for the CAIS. This technique constructs test cases in a White-Box testing fashion from an Abstract Machine description of the CAIS. The tests are administered in a Black-Box testing fashion. The Operational Definition of the CAIS (CAISOD) serves as input to the White-box testing method.

Symbolic execution is used to analyze the code to identify validation tests based on execution paths through the operational definition. The test paths are represented by input/output (I/O) pairs which consist of a range over the input interface parameters and a list of corresponding altered outputs for each execution path [5]. These I/O pairs are then converted into validation tests. Coleman [3] further refines the method described by [5] for generating a CAIS validation test suite based on symbolic execution. The method requires a valid Ada-based Abstract Machine description (operational definition) of the CAIS.

This thesis discusses the detailed design and preliminary implementation of generating a more complete set of test cases using symbolic execution. A general overview of the symbolic execution technique as it applies to testing is presented in Chapter 2. Chapter 3 presents the design of the I/O generator (IOGEN) and the approach taken to automate the validation method described by [5] and [3]. A case study is presented in Chapter 4 demonstrating the technique described in Chapter 3. The limitations of and possible extensions to IOGEN are discussed briefly in Chapter 5. Finally, Chapter 6 provides some conclusions about the thesis.

CHAPTER 2

Symbolic Execution

2.1. General Overview

Software development is largely the process of communicating information about the eventual program and translating this information from one form to another [9]. Verifying that the end product of this translation process, the computer program, behaves according to its specifications, is the focus of attention of many research groups. Symbolic execution, one method of program verification, is the approach taken by Hantler and King [6] for proving programs correct.

Hantler and King [6] provide a method utilizing assertions to verify the correc. An input assertion, represented as an ASSUME statement and inserted at the beginning of a routine, places constraints on all inputs for the routine. An output assertion, represented as a PROVE statement and inserted immediately before the return from a routine, represents the expected relation between the inputs and outputs. The routine is said to be correct if the truth of the input assertion guarantees the truth of the output assertion.

The proof for correctness for any given program is a proof for all possible program inputs, not just a subset of these inputs. Hantler and King [6] suggests using symbolic values to represent arbitrary program inputs. By doing so, numeric variables take on "symbolic" as well as numeric values. These symbolic

values can be represented as an elementary symbolic value, an arbitrary string chosen to represent a variable, or an expression in numbers and arithmetic operators. In this thesis, symbolic values are represented as Greek letters.

Before symbolically executing a routine, all input parameters are assigned a unique symbolic value. Symbolic execution continues by substituting all occurrences of the input parameter on the right hand side of an assignment statement by its symbolic value. The result isan algebraic equivalent of the numeric value.

Symbolic execution of routines containing iterative and selective constructs results in branches in the execution tree. Each path through the tree identifies an execution path through the program. Attached to each path is a predicate, called a path conditions (pc), describing the conditions which cause a path to be executed. The pc receives an initial value of true at the beginning of symbolic execution for all routines. As branches are encountered in the symbolic execution tree, the pc is modified to reflect the path condition for each branch by using an AND operation. The following sections discuss input/output (I/O) pair generation using symbolic execution for several Ada constructs.

2.2 I/O Pairs Generation using Symbolic Execution

We assume that the CAISOD, which serves as the input for symbolic execution and as the specification, has been validated by some other means [3]. Symbolic execution as presented by Hantler and King [6] is used as a means of

generating I/O pairs, not as a means of proving a program's correctness. Modifications to the method are made to provide I/O pair generation. Since the input assertion at the beginning of any routine is always true, it has been eliminated. Additionally, because the operational definition is assumed to be correct, the PROVE statement is also eliminated.

The execution trees generated as a result of symbolic execution are used to devise a set of I/O pairs. The input portion of the I/O pair is produced by examining the path conditions at the bottom of each execution path within the execution tree. The path condition can be reduced, thereby yielding an algebraic expression representing a "value" which causes that path to be executed. All symbolic values in the resulting algebraic expression are substituted by their corresponding global, local, or parameter variables in order to generate test data for the routine. The resulting data may then be used to establish initial values for global variables or values for the input parameters.

The output portion of the I/O pair results from actions taken along a path within the execution tree. The actions taken represent modifications of global variables and output parameters. The following sections discuss symbolic execution for several Ada programming constructs.

2.3 Assignment Statements

Assignment statements are normally executed by replacing the variables in the right-side expression by their corresponding numeric value. The indicated

operations are then performed and the result is then assigned to the left-side variable. Symbolic execution replaces the variables in the right-side expression with a symbolic value. Because the right-side expression consists of an algebraic expression containing symbols and not numeric values, it is left unchanged. This algebraic expression then become the new value for the left-side variable.

2.4 If-Then-Else Statements

Symbolic execution of If-Then-Else statements is similar to the normal execution for this conditional branching statement. First, all variables in the boolean expression are replaced by their corresponding symbolic value. Two separate expressions are formed using the new boolean expression. One expression represents a true boolean condition. The second expression represents the negated (false) boolean condition.

A conditional branching statement such as If-Then-Else causes a fork in the execution tree. One path represents the true boolean expression or *then* path. The other path represent the false boolean condition or *else* path. These paths do not rejoin at a point later in the execution tree.

Path conditions for the two separate paths are formed by ANDing the current pc with each boolean expression formed earlier. The *then* path's pc is formed by ANDing the current pc with the true boolean expression. The remaining path's pc is formed by ANDing the current pc with the false boolean expression. This represents the *else* path. If the statement does not contain an

else clause, modifications to the pc are not necessary. Execution for both paths continues with the instructions following the If-Then-Else statement.

```
function  VOWELS  ( char :  in character ) return integer is

type  vowel_char is ('A','E','I','O','U');
temp   : integer;
vowel :  vowel_char;

[1]  begin

[2]        if  ( char in vowel )  then
[3]             temp := -1;
[4]        else
[5]             temp := 0;
[6]        end if;
[7]        return temp;
[8]     end;
```

FIGURE 1. Function VOWELS

Figure 1 presents an example of a function which returns a integer value (0 or 1) indicating whether the input character is a vowel. The symbolic execution tree for the corresponding function is presented in Figure 2. The symbolic value "π" represents the variable *char*. Figure 2 presents an example of symbolic execution for the conditional branching statement and shows the separation of paths with their corresponding path conditions.

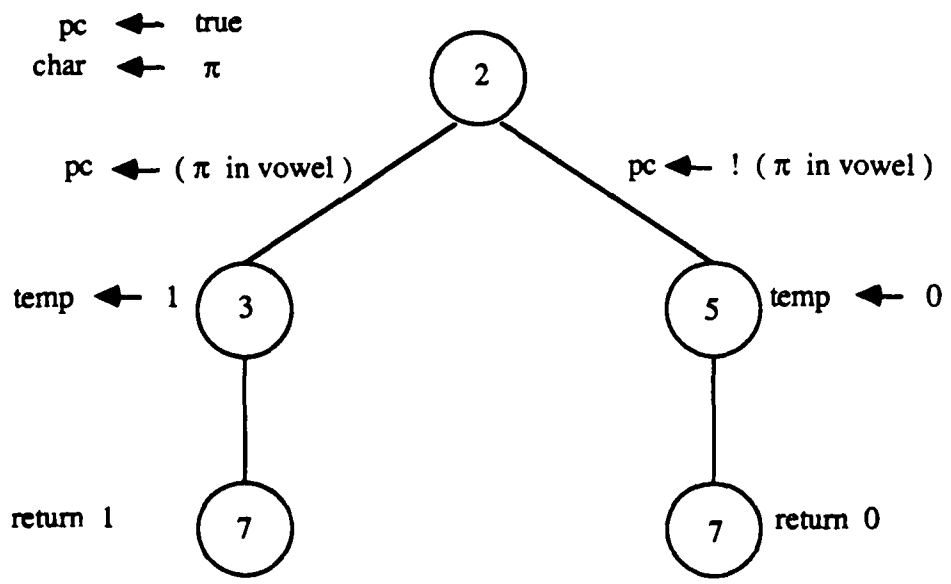


FIGURE 2. Symbolic execution tree: Function VOWELS

2.5 Looping Constructs

Introducing loops into a routine implies that the symbolic execution trees corresponding to the routine can be infinite [8]. It is obvious that routines which have non-terminating loops have infinite execution trees. For routines which do terminate, the size of the execution tree are finite, but can be exceedingly large. Substituting symbolic values for actual variables during symbolic execution

represents a problem when looping constructs are involved since a unique symbolic value must be generated for each actual variable.

To solve this problem, Hantler and King [6] use a form of induction to prove programs correct. Essentially, by focusing attention to only a finite section of the symbolic execution tree, the problem can be eliminated. This is accomplished by inserting inductive assertions into the symbolic execution tree, thereby restricting attention to a finite portion of the tree.

Since the operational definition of the CAIS is assumed to be correct, verifican of loop correctness is not necessary. The focus of attention, however, must be narrowed. To accurately reflect the loop construct when generating I/O pairs, the number of loop iterations must be determined. Generally, this number is not known prior to execution. In this case, a single iteration is considered. Additional tests may be necessary if it is determined that a single iteration is not sufficient.

The Ada looping constructs For, While, and Exit-When are examined. Symbolic execution for these constructs is similar to symbolic execution of If-Then-Else statements. A fork in the execution tree is created with one branch representing a path around the loop, a false loop condition. The pc for this path is determined by ANDing the current pc with the false loop condition. The other path represents a single iteration of the loop whose pc is determined by ANDing the current pc with the true loop condition. Once this path has been executed, the

loop condition becomes false and must be reflected accurately in the path condition. This is done by ANDing the pc with a false loop condition.

Consider the procedure COUNT_VOWELS in Figure 3. COUNT_VOWELS counts the number of vowels appearing in a string. COUNT_VOWELS calls the function VOWELS which appeared in Figure 1. Symbolic execution of the routine COUNT_VOWELS results in the execution tree presented in Figure 4.

```
type letters is 'A' .. 'Z';
type char_string is array <> of letters;
index : integer;

function COUNT_VOWELS (
    length : integer;
    instr  : char_string ) return integer is

vowel_count : integer := 0;
temp        : boolean;

  [1]  begin

  [2]  while index <= length loop
  [3]      temp := VOWEL(instr(index));
  [4]      vowel_count := vowel_count + temp;
  [5]  end loop;

  [6]  return vowel_count;

  end VOWEL_COUNTS;
```

FIGURE 3. Function COUNT_VOWELS

The example given in Figure 4 demonstrates symbolic execution for the

While loop. Symbolic execution for the For loop is identical to the While loop. One branch in the tree represent an out-of-range index when the For loop is

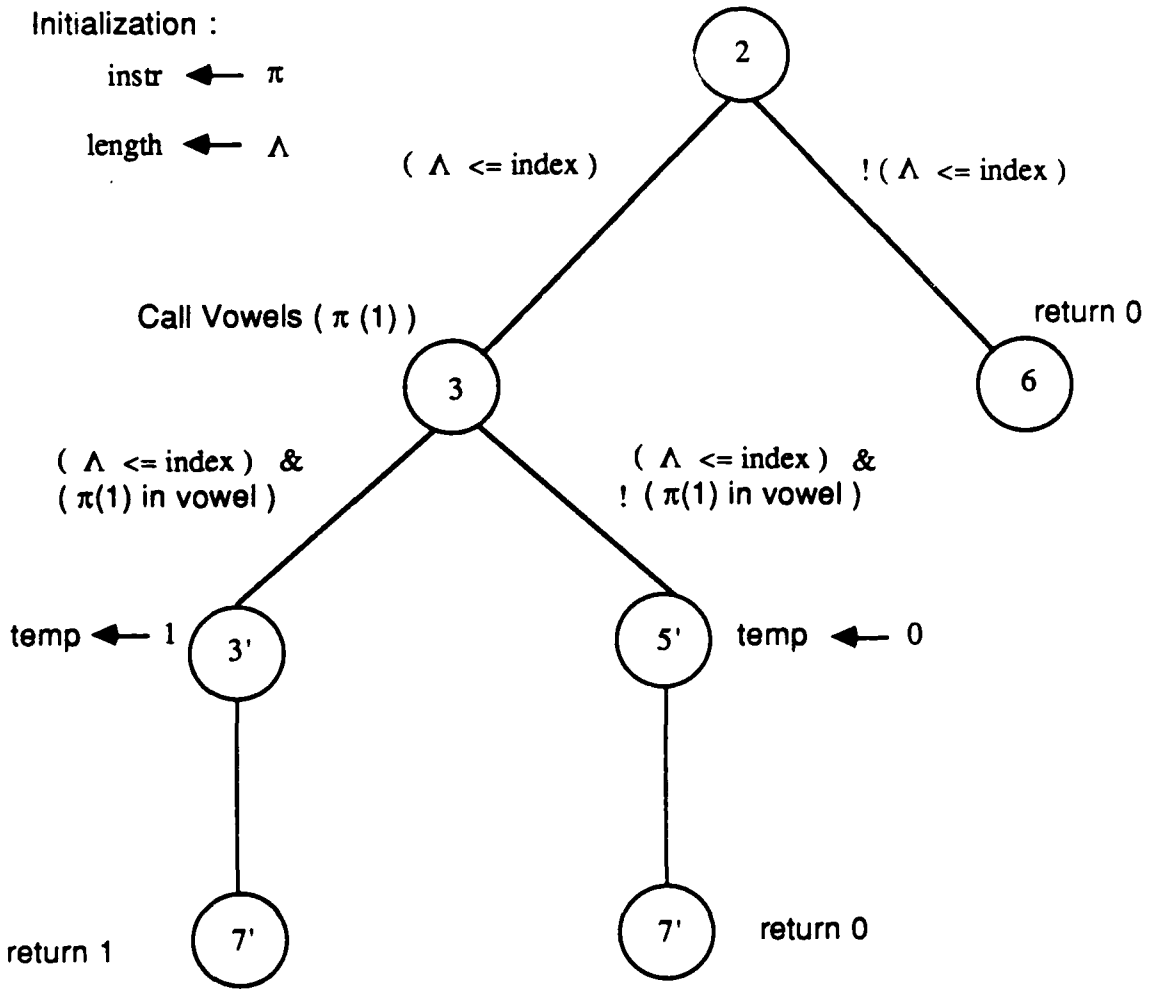


FIGURE 4. Symbolic execution tree: Function COUNT_VOWELS

encountered while the other branch represent a single iteration followed by an out-of-range index. There is a slight variation in symbolic execution for the EXIT-WHEN looping construct. A fork is generated in the execution tree when the EXIT-WHEN condition is encountered. The exit condition is met the first time it is encountered for one branch. The boolean condition is assumed to be false for the second branch, therefore execution continues with the statements following the EXIT-WHEN statement. For each branch, execution continues with the statements following the looping construct.

2.6 Case Statement

The Case statement, like the If-Then-Else statement, allows for the selection of alternative paths of control. Unlike the If-Then-Else statement which allows only two alternative paths, the Case statement all for *N* possible paths. If there are *N* possible path of control, the fork in the execution tree has *N* branches. Each branch's path condition, except for the *others* choice, is determined by ANDing the current pc and the expression *case_selector = choice*. The pc for the *others* choice causes the current pc to be ANDed with the negation of every possible path. Execution for each branch continues with the statements following the selected case and then the statements following the Case statement. . ure 6 shows the execution tree for a Case statement in Figure 5.

```
[1]   case COUNT is

[2]         when 1   =>   temp := 1;
[3]         when 2   =>   temp := 2;
[4]         when 3   =>   temp := 0;

[5]   end case;
```

FIGURE 5. Case statement

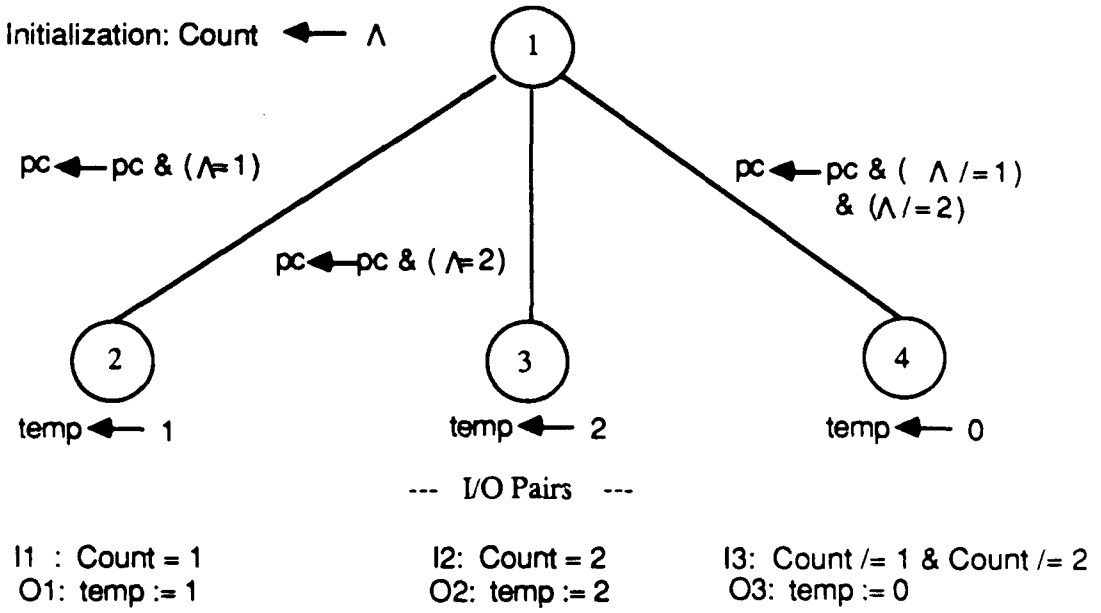


FIGURE 6. Symbolic execution tree: Case statement

2.7 Procedure Calls

The transfer of control caused by a procedure or function invocation can be approached in two different ways when generating the symbolic execution tree.

approached in two different ways when generating the symbolic execution tree. One approach is to continue execution as if the procedure was contained within the main routine. This involves a macro-like expansion into the symbolic execution tree [5]. Caution must be taken not to confuse local variables with the variables of the calling routine. In fact, this method requires a "start from scratch" approach because it causes the procedure code to be retested with every invocation.

The approach designed by [5] suggests that the symbolic execution tree be built in a bottom-up fashion. The I/O pairs for the called procedure are generated prior to the calling program's invocation. The I/O pairs generated are expressed in terms formal parameters. If there are *N* I/O pairs corresponding to the called procedure, an *N*-way branch in the execution tree is generated. The current pc of the calling procedure is joined with the input portion of each I/O pair by an AND operation to generate a unique pc for each branch. The new pc generated for each branch should be expressed in terms of the symbolic values corresponding to the actual parameters. Therefore, the formal parameters are substituted by their corresponding symbolic value.

Figure 4 presents the symbolic execution tree for the function COUNT_VOWELS. Figure 2 illustrates the I/O pairs necessary for the procedure COUNT_VOWELS. The two I/O pairs generated cause a 2-way fork in the execution tree.

The path conditions resulting from the AND operation of the calling program's pc and the I/O pair generated by the function VOWELS are:

- (Δ <= length) & (π in vowel)
- (Δ <= length) & ! (π in vowel)

2.8 Summary

This chapter presented an overview of the symbolic execution technique designed by Hantler and King [6]. This technique was modified and applied to several Ada constructs in order to construct symbolic execution trees representing these constructs. Using the execution trees produced, input/output pairs were generated.

CHAPTER 3

Input/Output Pair Generation (IOGEN)

The detailed design of the automated system which generates I/O pairs is presented in this chapter. The inputs required, outputs generated, and the structures used in the system are discussed. The major components of the system are also discussed.

3.1 Input and Output for IOGEN

IOGEN, an automated system for generating I/O pairs, symbolically executes source code from the CAISOD and produces a set of I/O pairs representing the possible execution paths through a routine. In order to generate the I/O pairs, IOGEN requires the name of two files as input. The first file contains the CAISOD source code which is written in Ada. The set of I/O pairs generated by IOGEN is directed to the second file specified.

The CAISOD source code consists of a separate package specification and body. Because the type declarations and forward declarations of procedures and functions contained with the package specification do not play a role in the symbolic execution process, the specification for any package is ignored. IOGEN therefore expects a package body as input. All global number and object declarations placed in the package specification must be relocated from the package specification to the package body by the user. It is important that this be

accomplished because IOGEN must generate a structure for all global variables.

3.2 Data Structures

The execution trees produced by the symbolic execution method presented in the previous chapter contains 2 and/or 3 way branches. Figure 7, for example, represents a tree which could be generated by IOGEN.

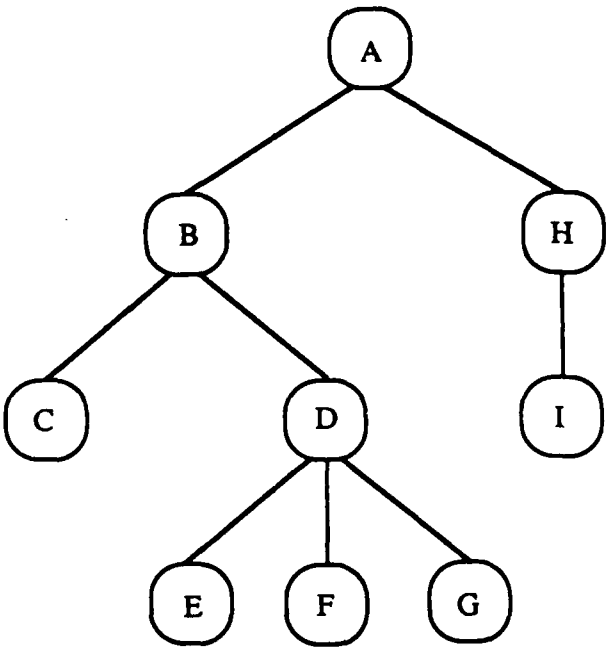


FIGURE 7. Tree

Each node in the execution tree demonstrates some action taken when a particular Ada construct was encountered. Each path condition represents the initial state causing that particular path te executed. Nodes A and B represent an Ada construct which causes a fork in the execution tree, while node D represents

either a case statement with three alternative paths or a procedure call which generated three distinct I/O pairs. Nodes C, E, F, G, H, and I represent nonbranching constructs such as assignment statements. While this is just a pictorial version of a tree, a symbolic execution tree created by IOGEN is based on nodes and pointers. The implementation of the tree shown in Figure 7 is presented in Figure 8.

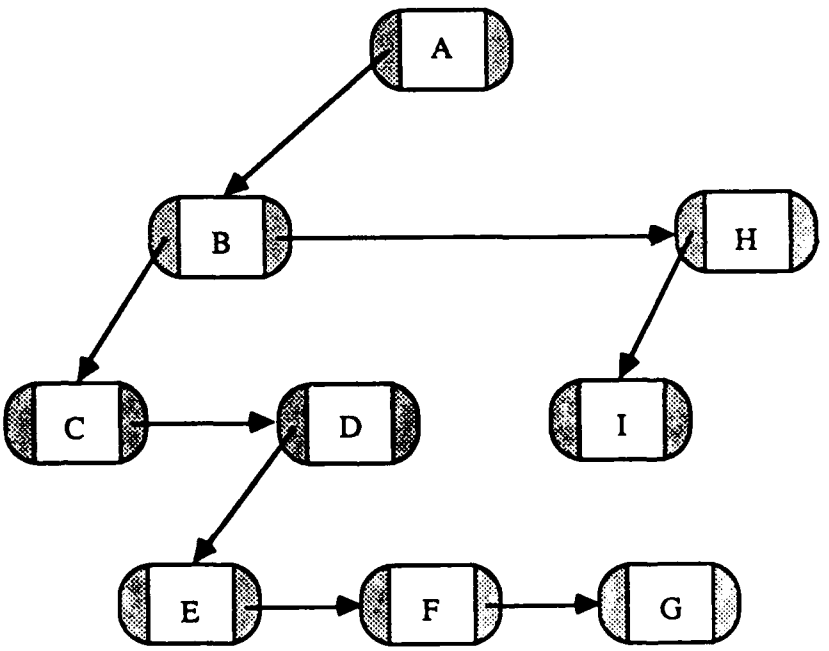


FIGURE 8. IOGEN's symbolic execution tree

The actions taken along a particular path are represented in one of two information fields as indicated in Figure 9. The connections between nodes as indicated in Figure 7 represent path conditions, while the pointers in Figure 8

represent actual links or addresses to succeeding nodes within the execution tree. In order to represent the path conditions between nodes, an additional information field within the node was required. For example, the pc between nodes A and H in Figure 8 is represented in the *in_ptr* field of node H. The path condition and action taken are by no means the only information contained within the node. Additional information which eases the implementation effort is contained in the node and is discussed in the detailed design section later in this chapter.

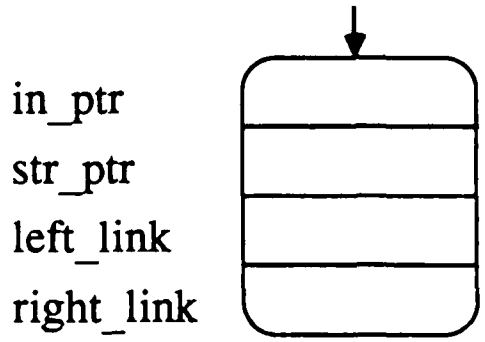


FIGURE 9. IOGEN Node

3.3 Major Components of IOGEN

Four distinct components for the IOGEN system were necessary in order to generate an execution tree and a set of I/O pairs corresponding to the execution tree: a scanner, a parser generator, a parser, and an I/O pair generator.

3.3.1 Scanner

When IOGEN accepts as input the file containing a package body, it is

viewed as one long stream of characters which must be assembled into a format more suitable for manipulation. The primary purpose of the scanner, or lexical analyzer, is to convert this unstructured stream of characters into a set of terminal symbols or tokens. The basic lexical units which IOGEN's scanner, called Get_a_token, accepts are keywords such as *package* and *loop* , symbols such as < , /= , and (, and identifiers. All keywords, symbols, and identifiers recognized by the Ada language are considered as valid tokens by IOGEN.

The basic function of IOGEN's scanner is to convert the input characters to tokens (terminal symbols). The text is examined character-by-character to determine whether the symbol is an extraneous blank, a symbol continuing a previously seen token, or the beginning of a new token. The action appropriate to the type of symbol recognized is taken.

Two additional functions are performed by the scanner. First, comments and extraneous blanks are removed. Second, the scanner must eliminate the *type* and *subtype* declarations which can appear in procedures and functions defined in the package body. These declarations are irrelevant in generating a symbolic execution tree. It is unnecessary for IOGEN to perform error detection since only previously compiled code is accepted as input.

One pass over the text provided in the input file is made. As the pass proceeds, tokens are provided to the parser upon request. This is done one token at a time.

3.3.2 Parser Generator

RRIPLL (pronounced as ripple), a parser generator tool developed at Arizona State University, builds a major component of the IOGEN system, its parser. RRIPLL requires as input:

- an LL(1) grammar
- the name of the scanner (Get_a_token)
- a list of terminal symbols returned by the scanner
- a list of non-terminal symbols

Every programming language has a set of rules characterizing the correct form of programs for that language [10]. This set of rules determining the format of programs is known as a grammar. Grammar rules consist of left and right-hand sides. The left-hand side, a unique non-terminal, denotes the name of the grammar rule (syntactic category) and is defined to be the sequence on the right-hand side. The right-hand side consists of a sequence of zero or more non-terminal and terminal symbols. An example of a grammar rule for the non-terminal *package_body* is shown in Figure 10. An equivalent syntax graph for the non-terminal is shown in Figure 11. Syntax diagrams for the grammar rules provided to RRIPLL are shown in Appendix A [2]. The rules are a subset of the Ada language.

Using the previously mentioned inputs, RRIPLL produces a predictive, top-down parser. This table-based parser, written in Pascal, deterministically

```
package_body ::=
        PACKAGE BODY package_simple_name IS
      [          declarative_part          ]
      [    BEGIN  sequence_of_statements    ]
      END     [     package_simple_name     ]
```

FIGURE 10. Grammar rule : Package_body

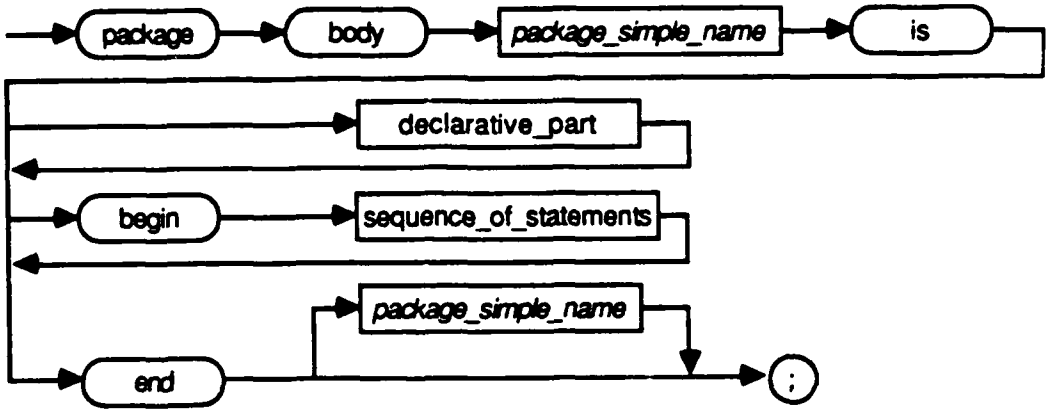


FIGURE 11. Syntax diagram : Package_body

parses a string from left to right with a single token of lookahead. That is, it parses without making an incorrect choice as to which grammar rule, or production, to apply next. Given a choice of two alternative productions, the next token indicates

the one and only choice of productions that leads to a successful parse.

3.3.3 Parser

The focal point of the IOGEN system is the parser produced by RRIPLL. Its basic structure is that of a table driven recursive descent parser (a predictive parser). Procedure calls which invoke various external Ada routines are permitted by RRIPLL and are integrated into the parser.

A parser generally accepts the output generated by the scanner and verifies that the source program satisfies the grammatical rules of a language [10]. Various structures such as parse trees and symbol tables are created as outputs. The primary purpose of the parser in IOGEN, however, is to generate a symbolic execution tree. The parser accepts tokens produced by Get_a_token and invokes the appropriate Ada action routines. These action routines build node structures, which contain information indicative of the token and statement type. At various points during the parsing stage, these nodes are inserted into their proper locations in the symbolic execution tree.

3.3.4 I/O Pair Generation

Upon completion of the parsing phase, a symbolic execution tree representing the execution paths within the source routine has been created. Traversing the symbolic execution tree creates the I/O pairs. Examining the nodes within the tree determines whether the node represents an action (i.e., assignment) or a change in path condition. If the node represents an action node,

the information contained within the node is appended to the output portion of the I/O pair. If it represents a condition node, the stored information is inserted into the input portion of the I/O pair. Once the entire symbolic execution tree has been traversed, all I/O pairs generated are outputted to the output file designated by the user.

3.4 Detailed Design of IOGEN

The objectives of the IOGEN system were two-fold. First, IOGEN accepts as input CAISOD routines and generates from these routines I/O pairs. Second, all routines necessary for generating a symbolic execution tree were to be written in Ada. With one minor exception, these objectives were achieved. The parser produced by RRIPLL is Pascal, not Ada code. If a compiler-compiler producing Ada code becomes available, the parser could be re-generated; or with minor effort, the Pascal code produced by RRIPLL could be converted into Ada. However, the routines required to produce the symbolic execution tree and I/O pairs are written in Ada and are external to the Pascal parser.

As previously mentioned, the acceptable input to the IOGEN system is an Ada package body. The three components of the package body, as shown in Figure 12 are:

- global declarations
- global procedure/function declarations
- body

```
PACKAGE BODY  name IS
     [        global declarations        ]
     [  global procedures/functions      ]
     [ BEGIN
          sequence_of_statements  ]
END [              name                 ];
```

FIGURE 12. Package body

Although these components are optional within a package body, the body of the package itself is required by IOGEN. The processing of these components is discussed in detail in the following sections.

3.4.1 Global Declarations

Global declarations for the package body consists of the following:

- number declarations
- object declarations
- *type* declarations
- *subtype* declarations

Number declarations create objects of a constant numeric value and may be any subset of the *integer* or *float* predefined type. The following are valid number declarations:

```
month : constant := MAY;
```

pi : constant := 3.14159265;

Object declarations create variables of a predefined or user-defined type. IOGEN, however, does not recognize subranges for predefined types. The parser recognizes type definitions only. For example:

bottom : INTEGER;

is a valid variable declaration while

bottom : INTEGER range -10 .. -1;

is not. Note that constrained array definitions are not permitted when defining variables. The constrained array construct is not included in IOGEN's grammar. Default values in variable declarations are permitted if not of the predefined type *string*.

If the lexical element represents a number or object declaration, an object of type ID_NODE is created. An example of ID_NODE is shown in Figure 13.

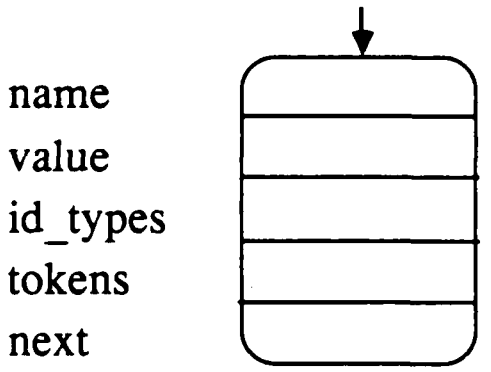


FIGURE 13. ID_NODE

The information field *name* is a string whose value is the identifier's name. If a default value has been assigned, it is contained in *value*, otherwise, the value assigned is "undefined." *Id_type* is used to distinguish whether the identifier is located in the package body or subprogram. If the identifier was defined in the package body, the value of *id_type* is GLOBAL_ID, otherwise, LOCAL_ID is assigned. *Tokens* contains the address to a list of tokens representing the value of the object. Additional information on *tokens* is provided in section 3.4.3.1. As each object is created, it is inserted into a linked list structure which is maintained for the package body or current subprogram.

Type and subtype declarations are permitted if they span less then 81 characters and are contained in one input line. Type and subtype declarations spanning multiple lines are considered to be invalid input. As previously mentioned, these declarations are eliminated by the scanner.

3.4.2 Global Procedure/Function Declarations

Symbolic execution of source code containing procedure and function invocations requires as input all I/O pairs for the called routines. This is accomplished by generating a separate symbolic execution tree for each subprogram defined.

Subprograms have the following format:

Subprogram_specification is

{ local type declarations }

begin

sequence_of_statements

end { name };

For each subprogram encountered, an object, called PROC_NODE, is created. A linked list of these objects is maintained and referred to as subroutines are encountered in the main body of the package

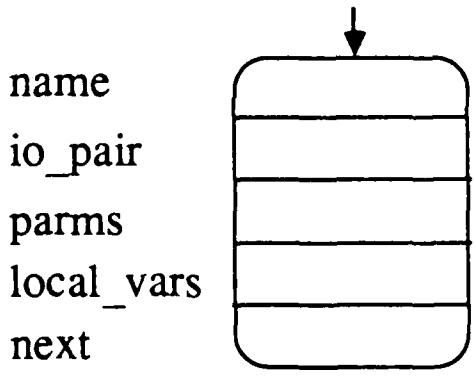


FIGURE 14. PROC_NODE

As shown in Figure 14, three information fields within the node are of particular interest; *io_pair*, *parms*, and *local_vars*. After each subroutine's execution tree has been generated, a traversal of the tree is performed to generate a set of I/O pairs for that routine. The address of the set of I/O pairs is maintained in *io_pair*.

Since only one pass is made over the input, information such as formal parameters and local variables must be preserved so it can be retrieved at a later time. Although local variables are not of major importance, their affect on formal parameters and global variables must be considered.

The generation of each subprogram's I/O pairs relies on the current value of formal *in* parameters and local variables. Therefore, maintaining accurate lists of parameters and variables is essential. In order to accomplish this, two additional fields within PROC_NODE were generated, *parms* and *local_vars*. *Parms* contains the address for a list of ID_NODEs whose current *id_type* value indicates whether the parameter was an *in* or *out* parameter (see Figure 15).

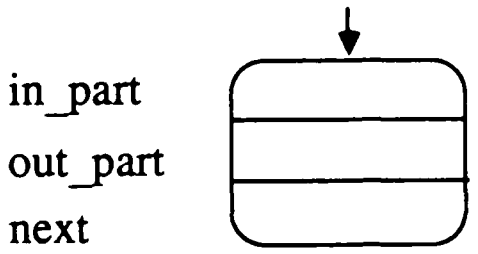


FIGURE 15. IO_NODE

This distinction must be made since I/O pairs should reflect changes in *out* and *in out* parameters but not in *in* parameters. Because Ada functions do not permit parameter types other than *in*, this does not apply. The current implementation of the system does not permit assignment of default values to formal parameters, therefore the*value* field in ID_NODES is always undefined. *Local_vars* is the

address a list of ID_NODEs created for each variable defined within the subprogram. Each node may contain an initial default value and is assigned an *id_type* of LOCAL_ID.

Once the subprogram's symbolic execution tree has been generated, it is traversed to determine which actions affect*out* and*in out* parameters. This is accomplished by examining each node's *identifier* element and comparing its value with elements in the *parms* and*local_vars* list. Actions modifying local variables are ignored. Actions modifying *out* and *in out* parameters are recorded in the appropriate I/O pair. If the *identifier* element is not located in either list, the variable represents a global identifier and is also recorded in an I/O pair.

3.4.3 Body

The body of a package consists of a sequence of statements, simple and compound, which forms the primary symbolic execution tree in the IOGEN system. Each statement is examined to determine whether it represents an action or condition which causes a fork within the execution tree. A node is generated accordingly.

The following statement types are examined in the following sections:

- Assignment statements
- Procedure/function call statements
- Return statements

- If-Then-Else statements
- Case statements
- Exit statements
- Loops

3.4.3.1 Assignment Statements

IOGEN accepts any legal Ada assignment statement with an expression consisting of variables and operators defined by the Ada language. As lexical units are encountered in the right-side expression, IOGEN determines whether the token represents a variable, symbol, operator, or numeric value. A TOKEN_NODE is created for each token and a type is assigned accordingly (see Figure 16).

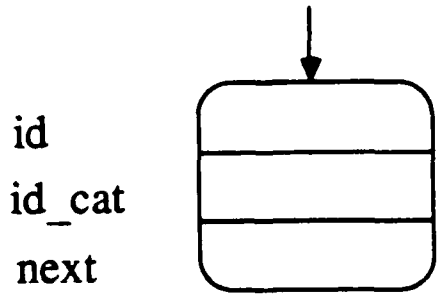


FIGURE 16. TOKEN_NODE

After the entire execution tree has been built, each token within a node's *tokens* list is evaluated. Tokens representing symbols and operators are used by IOGEN for grammatical analysis only and require no further action. If the token

represents a variable, the global variables list is searched to determine its current value. The token within the expression is then substituted by the variable's current value. If the assignment statement occurs within the body of a subprogram, the list of parameters and local variables is searched prior to the list of global variables. If the token represents a numeric value, no substitution is made.

Once the entire expression has been evaluated and all necessary substitutions have been performed, the object of assignment is located in the appropriate list and its value is then updated.

The action node representing the assignment statement has a designated *stmt_type* of "asm". The node field *token_list* contains the address of a list of tokens which make up the assignment statement. This list is used when the individual tokens of the statement must be examined. A node field is provided which allows a string to represent the original assignment statement. Additionally, the individual token representing the object of assignment is contained in a node field. This allows for quick referencing of the identifier in a given node.

3.4.3.2 Procedure/Function Call Statements

Procedure and function calls have the same effect on the symbolic execution tree being generated and require several actions to take place. The PROC_NODE for the corresponding subprogram is located and the number of I/O pairs must be determined. An N-way fork in the execution tree, which reflects the

number of I/O pairs generated within the subprogram, must be created. For each node in the tree fork, the path condition must reflect actual, not formal parameters. Therefore, substitutions between actual and formal parameters must occur. These substitutions must also occur within the output portion of the I/O pair. Once these substitutions have been performed, the output portion then becomes an action node along the indicated path. The address of this action node is retained in the *left_link* field of the path condition node. This process must be completed for each I/O pair represented in the PROC_NODE for that subprogram.

3.4.3.3 Return Statements

The return statement, which is permitted in functions only (by IOGEN), indicates a transfer of control back to the routine of invocation and the return of a value. Its structure,

RETURN *expression*;

may consist of any legal Ada expression. Return statements pose a serious problem when used throughout a function. During the generation of the function's execution tree, careful attention must be made not to append additional nodes to the node of a return statement. This would indicate that control was transferred yet processing within the function continued. To resolve this problem, the node is marked as a return node by designating the node's *stmt_type* as "retrn." Appending additional nodes to a "retrn" node is not permitted.

Once a function's execution tree is complete, a "retrn" node represents the

last node in each execution path. The absence of a return node indicates an unattainable transfer of control and the existence of an error.

3.4.3.4 If-Then-Else-Statements

If-statements, which must be of the form:

IF < *condition* > THEN

sequence_of_statements

{ ELSE

sequence_of_statements }

END IF;

generates a fork in the symbolic execution tree. Each fork node in the tree is of type SINGLE_NODE and represents a condition node. The *condition* represents any legal boolean expression. All tokens within the *condition* must be examined to determine whether substitutions are required. If so, the appropriate lists are searched and the proper substitutions are made. A string and a list of tokens is used to represent the altered If-statement's condition.

One node within the fork must reflect a true path condition. This is accomplished by retrieving the string representing the previously altered *condition* and assigning it to the information field of the node *in_ptr*. The address of the list of tokens representing the condition is stored in *cond_list*. The address of the sequence of statements which is executed when the path condition is true is stored in the node field *left_link*. This address represents the address of the

sub-tree generated as the sequence of statements is parsed. As a result, the sub-tree must be built prior to its address being inserted into the current symbolic execution tree, not while each statement is encountered during the parsing phase.

A node which represents the false path condition is generated regardless of whether an ELSE clause exists. If an ELSE clause exists, generating and storing the address of the sub-tree follows the same process as a true path condition. The path condition for the node is calculated by prefixing the *condition* string and tokens list with the character " ! ". The string and tokens list address is stored in the appropriate field.

The ELSE clause within the statement is optional, however, ELSIF clauses are not permitted by IOGEN.

3.4.3.5 Case Statements

Case statements take the form:

```
CASE selector IS

   WHEN choice => sequence_of_statements
END CASE;
```

The *selector* represents a valid Ada identifier while each *choice* consists of one

of the following:

- Identifier
- Discrete range
- Expression
- *Others* clause

The *selector* is represented by its symbolic value and each identifier within the various *choices* must reflect the current value of the identifier. Once again, substitutions between actual identifiers and their current value must take place.

When a case statement is encountered during the parsing phase, two separate lists are maintained. The first list represents a list of *choices* that is used if an *others* statement is encountered. Each *choice* within the list is negated and ANDed together in order to reflect an accurate *others* choice. The second list represents a list of SINGLE_NODEs, one for each*choice* within the case statement. Each node reflects an alternative execution path in the symbolic execution tree. Associated with each node is a list of action nodes representing the sequence of statements to be executed. The address corresponding to this list of nodes is maintained in the information field*left_link*. The *right_link* points to an alternate *choice* in the case statement.

3.4.3.6 Exit Statements

Exit statements must be of the form:

- EXIT WHEN < *boolean_expression* >

and indicates that a fork within the execution tree must be created. The nodes contained in the fork are of type SINGLE_NODE and represent condition nodes. The boolean expression within the statement is used to generate the path conditions for each node. The path condition for the first branch reflects a true boolean expression. A string representing the boolean expression is reflected in the node's *in_ptr* while the address of the list of tokens which form the expression is stored in *tokens*. Each variable within the*tokens* list is updated by its current value. Execution continues with the statements following the end of the loop in which the Exit statement is contained.

The remaining branch's path condition is generated by negating the expression. This expression is reflected in *in_ptr* as a string and *tokens* as a list of tokens. Execution continues with the statement following the EXIT-WHEN.

3.4.3.7 Loops

Three basic loop structures are examined:

-*For* loops

- *While* loops

- Simple loops

The *For* loop, which must be of the following format:

FOR < *loop_parameter_specification* > LOOP

sequence_of_statements .

END LOOP;

generates a fork in the execution tree. The conditions represented in each node of the fork consists of the *loop_parameter_specification.* One node represents a true specification, while its associated node's condition is negated. Once again, all variables contained within the specification reflect its current value.

The node representing the true *loop_parameter_specification* contains in its *left_link*, the address of the *sequence_of_statements*. This statement sequence is the address of a sub-tree which is created as the statements are parsed. As with all statement sequences parsed by IOGEN, the complete sub-tree is created prior to including its address in the appropriate node.

Once the sub-tree has been created and properly addressed, the *loop_parameter_specification* which represents the path condition must be negated to accurately reflect the termination of the *For* loop. In order to do so, the subtree must be traversed to determine the changes in each symbolic value contained within the specification. Because all variables do not have their symbolic values updated until the entire symbolic execution tree has been built, a node containing the negated specification is created. This node is appended to the bottom of each execution path of the sub-tree to reflect this change.

The node representing the negated *loop_parameter_specification* continues its execution with statements following the *For* loop.

The *While* loop, whose structure is:

WHILE < *condition* > LOOP

sequence_of_statements

END LOOP;

The sequence of steps taken to form a branch in the symbolic execution tree and the corresponding sub-tree are identical to the *For* loop. Note that instead of the *loop_parameter_specification* ,a boolean condition is used. The *condition* represents any legal boolean expression which is permitted by the Ada language.

Simple loop statements take the following format:

LOOP

sequence_of_statements

EXIT WHEN < *boolean_expression* >

sequence_of_statements

END LOOP;

Unlike the previous loop structures, a fork in the execution tree is NOT created when the loop is encountered. It is created when the EXIT-WHEN statement is encountered. The sequence of statements preceding the EXIT-WHEN is processed as usual. The sequence of statements following the EXIT-WHEN is processed in the tree branch representing a false boolean condition as described in the previous section.

3.5 Summary

This chapter discussed the different components of the IOGEN system and their basic function. A detailed description of how IOGEN processes various Ada constructs was also presented.

CHAPTER 4

Case Study

In this chapter, a case study of the technique provided in Chapter 3 for generating input/output pairs is presented. The data structures created by IOGEN as well as the symbolic execution trees representing sample routines are provided.

Coleman [3] presented two routines contained in the CAIS operational definition which converted a string of digits into its integer equivalent. These routines were examined and converted into a format acceptable to IOGEN. Minor changes were made for constructs and syntax not supported by IOGEN. The subprogram CONVERT was transformed from a function into a package body and the formal parameters were converted into a list of objects defined globally. The package body CONVERT is shown in Appendix B.

The declaration part of the package body consists of *type*, object, and function declarations. The*type* declarations are ignored by IOGEN but are included to provide better understanding of the package. Seven objects are defined: ACCUM, CHAR, I, SIZE, STR, TEMP, and VALUE. For each object defined, an ID_NODE is created. The resulting list of ID_NODEs is presented in Figure 17. The ID_NODE contains the name of the object, its default value, an *id_type* of GLOBAL_ID, and a pointer to the list of tokens representing the value

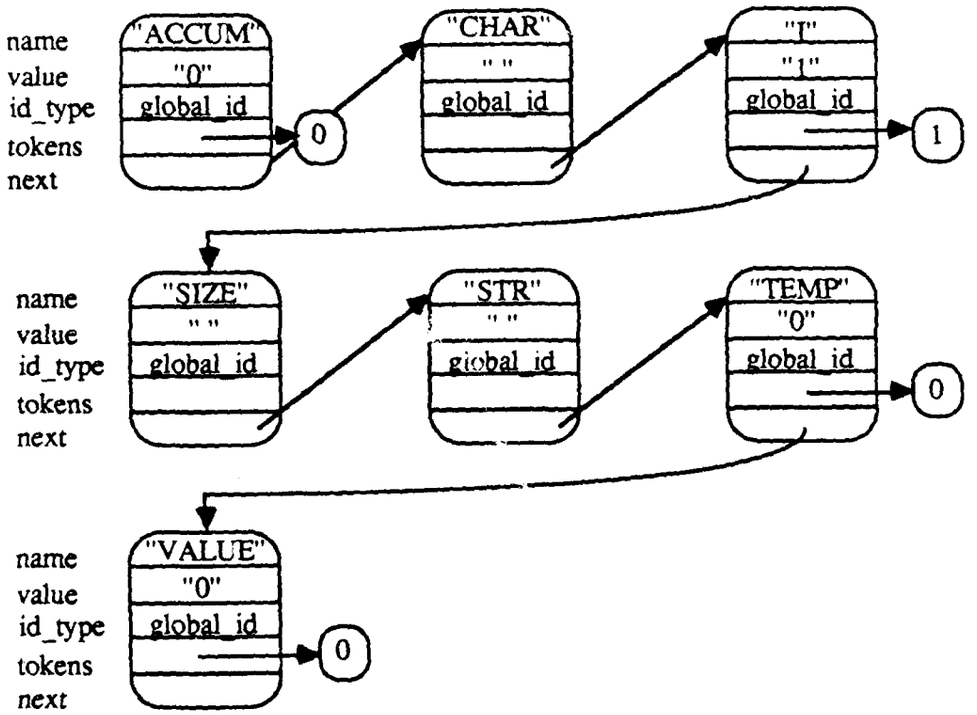


FIGURE 17. ID_NODEs for Package CONVERT

of the object. Default values are not assigned to CHAR, SIZE, and STR, therefore, *value* contains an empty string and *tokens* is assigned a null pointer. When the tree is traversed to generate I/O pairs, both the *value* and *tokens* field is updated to reflect the current value of the object.

The function ATOI is processed next. A PROC_NODE created for ATOI is shown in Figure 18. *Parms* contains a pointer to the only formal parameter

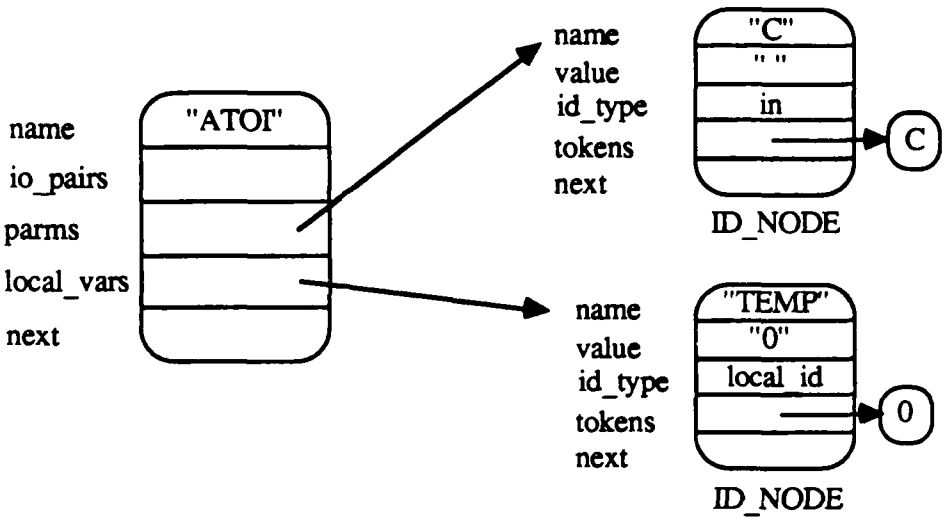


FIGURE 18. ATOI PROC_NODE

defined in ATOI, C. *Local_vars* contains the address of the ID_NODE created for the variable TEMP. A default value of zero has been assigned to TEMP. A pointer to the I/O pairs representing the paths within ATOI is stored in *io_pair*. At this point, *io_pair* is null.

The initial tree generated when the sequence of statements representing the body of ATOI is symbolically executed is presented in Figure 19. Several things should be noted at this point. First, if the node contains a value in the *in_ptr* field, the same value is represented in the *cond_list* field but in a different format. The same holds true for *str_ptr*. The same value represented in *str_ptr* is assigned to *token_list*. The Ada language lacks the complicated string manipulation functions needed by IOGEN. Therefore, *in_ptr* and *str_ptr*

represent the statement in its original form and *cond_list* and *token_list* represent the statement in a form which can be easily modified.

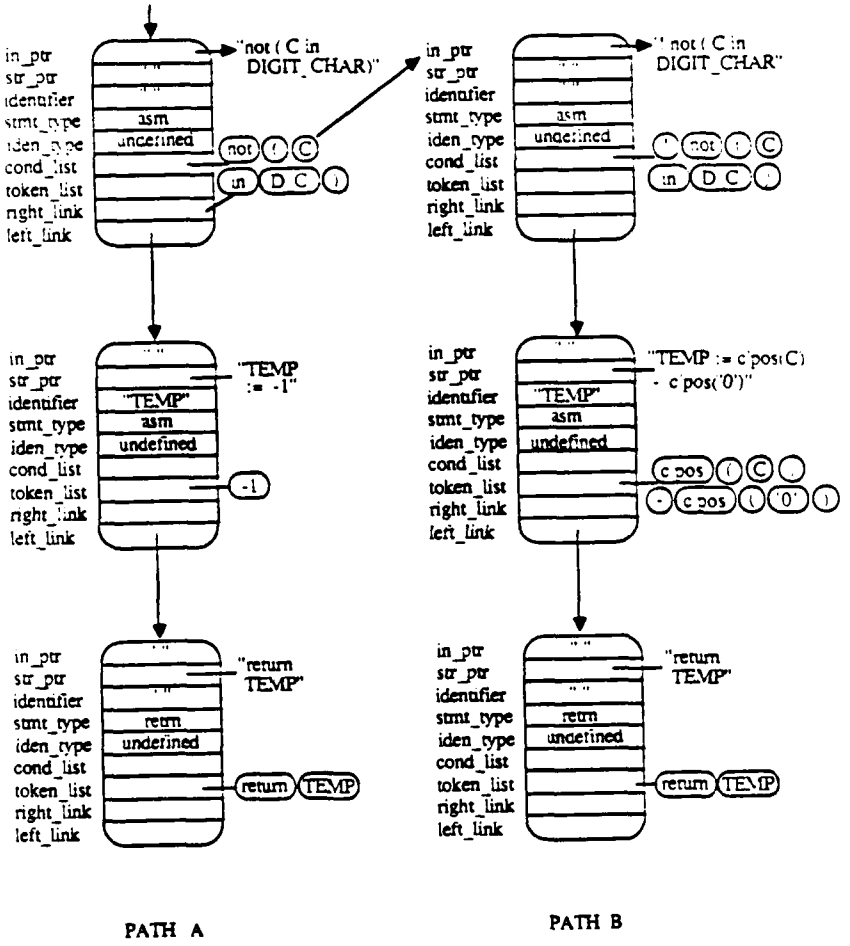


FIGURE 19. Symbolic execution tree: Function ATOI

The *stmt_type* field for these nodes can assume one of three values: proc, asm, or retrn. Nodes representing a return statement are assigned a *stmt_type* of "retrn". All other nodes have been assigned the value "asm". ATOI does not

contain subprogram invocations, therefore, the *stmt_type* "proc" has not been assigned.

Finally, for the two nodes representing the two assignment statements, the *identifier* field contains the name TEMP, the object of assignment. The *iden_type* field currently contains the value "undefined". Once the substitution process commences, this value is updated to LOCAL_ID to indicate that TEMP is a locally defined variable.

After the execution tree has been generated, a traversal of the tree is performed. All tokens representing identifiers within *token_list* and *cond_list* are substituted by their current value. To obtain an identifier's current value, two lists are searched, *parms* and *local_vars*. These lists are copies of the lists contained in the PROC_NODE for the function ATOI. If references are found in either list, the identifier is substituted by the list of tokens contained in the *tokens* field of the corresponding ID_NODE. If references are not found, the identifier is assumed to be global and no substitutions are made. The only substitution necessary in ATOI's execution tree is between TEMP and its current value. TEMP receives the value "-1" in the return statement of path A and "character'pos(C)-character'pos('0')" in path B.

If the node represents an assignment statement, *parms* and *local_vars* are searched once again to locate the variable referenced in the node's *identifier* field. The variable's value is updated by assigning the value of the node's

token_list to the variable's *tokens* field. For the function ATOI, TEMP's value is updated.

Finally, the I/O pairs for the procedure are generated. This is performed in conjunction with the previously mentioned actions. Each node is examined during the traversal to determine whether the information contained within that node should be included in the input/output pair. Only the assignment statements which modify global variables or *out* or *in out* parameters are reflected in the I/O pairs. All return statements are reflected as well as all compound statements. Once the traversal is complete, every path within the execution tree is represented. The input/output pairs generated for the function ATOI are:

I1: ! (C in DIGIT_CHAR)

O1: return -1

I2: (C in DIGIT_CHAR)

O2: return character'pos(C) - character'pos('0')

The *io_pairs* field in ATOI's PROC_NODE points to the I/O pairs generated above.

The final phase involves generating a symbolic execution tree for the body

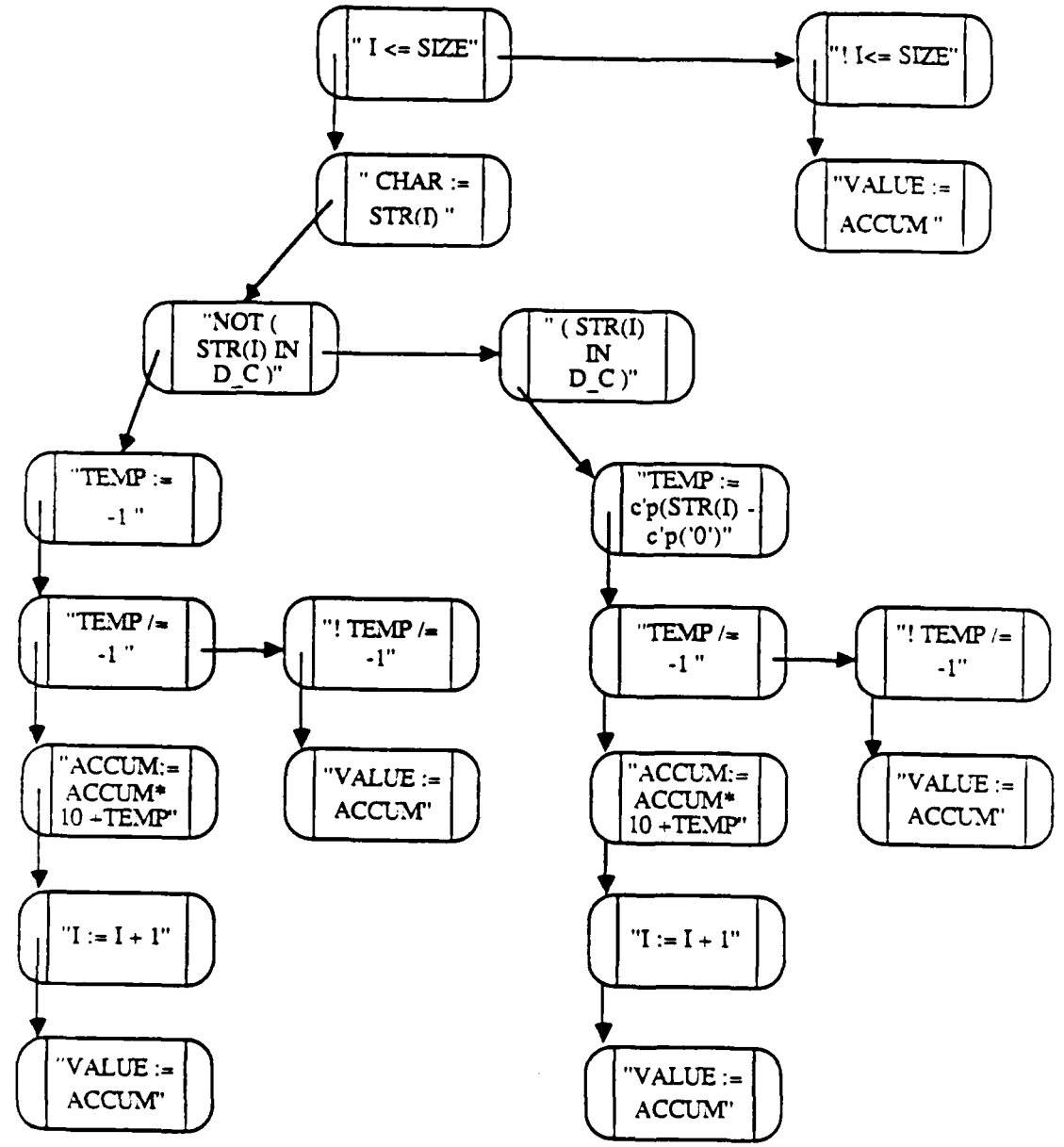


FIGURE 20. Symbolic execution tree: CONVERT

of the package CONVERT. The sequence of statements representing CONVERT's body produces the symbolic execution tree shown in Figure 20. The

only field shown is *in_ptr* or *str_ptr*.

During the generation of the execution tree for CONVERT, the statement

TEMP := ATOI (CHAR);

is encountered. Although similar in format to the statement

CHAR := STR (I);

the actions which take place differ greatly. A search for the variable STR in the global identifiers' list (see Figure 17) takes place and is successful. The statement therefore represents a simple assignment statement and an action node is created.

The assignment to TEMP, however, requires an additional list to be searched. The global identifiers' list is searched once again, but this time, the search is unsuccessful. The search continues by examining the list of PROC_NODES (see Figure 18). The identifier ATOI is located, thereby indicating a function call. The list of I/O pairs addressed by the pointer *io_pairs* in ATOI's PROC_NODE is examined to determine the number of I/O pairs generated. As previously shown, ATOI has two I/O pairs, therefore a two-way fork in the execution tree is generated. Each I/O pair generates two nodes within the tree. One node represent the input (path condition) while the second node represent the output associated with the input node. For example, the input/output pair

I1: not (C in DIGIT_CHAR)

O1: *return* -1

forms the structure shown in Figure 21.

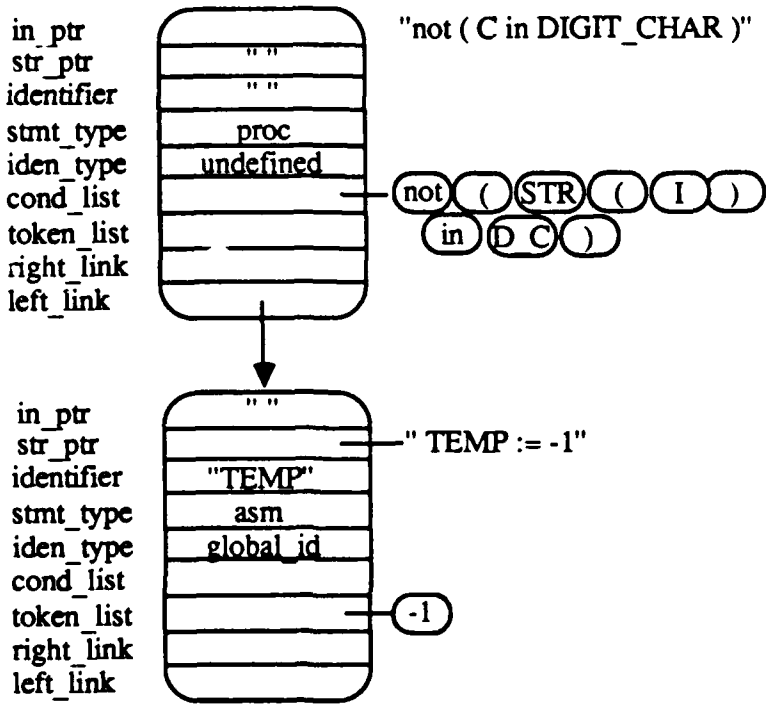


FIGURE 21. Input/output pair

Notice that the string in *in_ptr* differs from the list of tokens referenced by *cond_list*. This is caused by the substitution made between the actual and formal parameters.

After the symbolic execution tree has been generated for CONVERT, the tree is traversed. All necessary substitutions are made and all values are updated. Unlike the function ATOI which used the lists *parms* and *local_vars* to make substitutions, CONVERT uses the list of global identifiers referenced in the

ID_NODEs list.

Finally, the I/O pairs generated for the package CONVERT are:

I1: 1 <= SIZE & not (STR(1) in DIGIT_CHAR) & -1 /= -1

O1: CHAR: STR(1) & TEMP: -1 & ACCUM: 0 * 10 + -1 & I: 1 + 1 &
VALUE: 0 * 10 + -1

I2: 1 <= SIZE & not (STR(1) in DIGIT_CHAR) & ! -1 /= -1

O2: CHAR: STR(1) & TEMP: -1 & VALUE: 0

I3: 1 <= SIZE & ! not (STR(1) in DIGIT_CHAR) &
(character'pos(STR(1)) - character'pos('0')) /= -1

O3: CHAR: STR(1) & TEMP: character'pos(STR(1)) -
character'pos('0') & ACCUM: 0 * 10 + character'pos(STR(1)) -
character'pos('0') & I: 1 + 1 & VALUE: 0 * 10 +
character'pos(STR(1)) - character'pos('0')

I4: 1 <= SIZE & ! not (STR(1) in DIGIT_CHAR) &
! (character'pos(STR(1)) - character'pos('0')) /= -1

O4: CHAR: STR(1) & TEMP:character'pos(STR(1)) - character'pos('0') &
VALUE: 0

I5: ! 1 <= SIZE

O5: VALUE: 0

In summary, two symbolic execution trees were created, one representing the function ATOI and the second representing the package body CONVERT. The trees were traversed, all substitutions were made and I/O pairs were generated. Five I/O pairs were generated to represent the package CONVERT, one of which contained conflicting information. Chapter 5 addresses this issue along with other limitations of the IOGEN system.

CHAPTER 5

Limitations/Extensions to IOGEN

The preliminary design of the IOGEN system focused primarily on Ada branching constructs which generated forks within the symbolic execution tree for a CAIS routine. Other areas included in the design were:

- global declarations
- global procedure/function declarations
- package body

all contained within a package body. Restrictions, however, are present within each of these areas. This chapter focuses on the current limitations of the IOGEN system and discusses details for extending IOGEN in order to eliminate these limitations.

5.1 Global Declarations

IOGEN does not support global declarations such as *type* and *subtype* declarations which span more than one input line. It also does not support default values which are strings. In order to eliminate these restrictions, both the scanner and parser generator must be modified.

Currently, the scanner Get_a_token does not recognize the double quotes used to enclose a string as a special character. This special character could simply be added to the set of acceptable characters. The parser generator must

be modified to recognize not only the first double quotes signifying the beginning of the string but must also classify each following token as a continuation of the character string. Once a second double quote is recognized, the lookahead token must be examined. If the lookahead symbol is a semicolon, the end of the string has been encountered. If the lookahead symbol is another double quote, a double quote is contained within the string and parsing should continue.

To allow *type* and *subtype* declarations to span more than one input line, the parser generator must be modified. All valid *type* and *subtype* declarations must be included in the grammar. *Type* declarations which should be included are:

- full type declarations
- incomplete type declarations
- private type declarations.

In order to allow for multi-line *subtype* declarations, definitions for valid constraints, including

- discriminant constraints
- fixed point constraints
- floating point constraints
- index constraints
- range constraints

must be included in the grammar. Keep in mind that including these additional

declarations and constraints does not extend IOGEN's ability in handling the current statement types. This allows for fewer modifications to the original routine prior to submitting it as input to IOGEN.

5.2 Global Procedure/Function Declarations

IOGEN's limitation in procedure/function declarations include restrictions within the parameter list and subprogram body. Default values are not permitted in the formal parameter list of a procedure or function. To allow default values, the *parameter_association* grammar rule within the parser generator must be modified to reflect an optional production; *formal_parameter* followed by the symbol " =>". In addition, the initial default value for each parameter must be reflected in the ID_NODE for the corresponding parameter. If, during the substitution process, an identifier is located in the *parms* list, the *value* field must be examined to determine if an initial value for the parameter exists. This does not take place in the current system.

Currently, IOGEN does not permit subprogram calls from within a procedure or function. The reason is this. Forward declarations for procedures and functions are contained within the package specification which is not included as input. I/O pairs for a given procedure or function must be generated prior to the invocation of that subprogram. It is conceivable, however, that a call is made prior to the declaration of that subprogram. For example, procedures A, B, and C are defined in that order for a given package. Calls to procedure A from

within procedure B do not cause a problem. However, calls to procedure C from within procedure B cause a serious problem. The *N*-way branch representing a call to procedure C within procedure B's symbolic execution tree cannot be generated since the number of I/O pairs cannot be determined.

One possible solution is to maintain a list of addresses of nodes within the execution tree which require further expansion. If a call is made to a procedure which has not yet been defined, a node representing the call is included in the execution tree and the address of the node is inserted into the list of unexpanded nodes for that procedure. This process is repeated for each call to a subprogram not yet defined. Symbolic execution for the procedure continues with the remaining statements. If, however, the execution tree for a procedure is complete, all I/O pairs for that procedure are generated.

Once the execution trees for all subprograms have been defined, the PROC_NODE for each subprogram is examined to determine whether the list of addresses of unexpanded nodes is empty. No further action is required if the list is empty. If the list is not empty, nodes requiring further expansion must be examined to determine whether the I/O pairs for the corresponding subprogram have been generated. If the pairs have been generated, the unexpanded node is eliminated, an N-way branch is generated, and the tree is restructured. If the I/O pairs have not been generated for the subprogram, the called subprogram's execution tree must be examined and all nodes expanded. This procedure

continues until all trees have been fully expanded and all I/O pairs have been generated.

5.3 Package Body

Limitations contained within the body of the package are directed towards statement structures. Call statements do not allow explicit naming of parameters. If-statements do not permit *elsif* clauses and case statement selectors can only be identifiers. Note, these restrictions also apply to the body of subprograms.

In order to allow explicit naming of parameters in a procedure call, the *parameter_association* grammar rule within the LL(1) grammar provided to RRIPLL must be modified to reflect an optional production; *formal_parameter* followed by the symbol " => ". A comparison of names, whereas before, positions within the list, should be used to determine the proper substitutions between actual and formal parameters.

In order to extend the If-Then-Else statements so it includes the *elsif* clause, the *if_statement* production rule within the grammar must be modified. It should include an optional *elsif* production which states that the rule consists of the keyword "ELSIF", a boolean condition, and a sequence of statements. To represent the structure in the execution tree, an *N*-way branch must be generated. The path conditions for each *elsif* branch are constructed by ANDing the negation of each preceding path condition with the condition of the current path. The *else* branch's path condition is constructed in the same manner as the

others path in a case statement. All previous path conditions must be negated and joined by an AND operation.

To enable the selector of the case statement to take on values other than a single identifier, the *case_statement* production must be modified. By changing the token *id* to the production representing a *discrete expression*, the selector can represent an integer or enumeration type result.

5.4 Input/Output Pair Generation

The example given in the previous chapter demonstrates how I/O pairs for a routine are generated. Notice, however that the first pair generated contains conflicting information. The input portion indicates that the value of TEMP is not equal to -1 while the output portion states that the current value of TEMP is *equal* to -1. The value of TEMP is generated by a call to the function ATOI, in which a value of -1 is returned for invalid characters passed as parameters. IOGEN does not evaluate the information contained in each I/O pair to determine whether the pair represents a valid path within the symbolic execution tree. It is important to note that any I/O pair generated by IOGEN which contains conflicting information can be disregarded. It represents a path which is not executable due to the result of an expression or the value returned by a function or procedure. One possible area for further research is to design a method which prevents input/output pairs containing conflicting information from being generated.

Summary

The chapter discussed the limitations of the IOGEN system and recommended ways of modifying IOGEN in order to eliminate these restrictions. These modifications included altering both the scanner, Get_a_token, and the parser generator. Changes to the parser generator focused primarily on changes to the LL(1) grammar.

CHAPTER 6

CONCLUSION

A method using symbolic execution to construct the I/O pairs necessary for generating validation test cases for the Common APSE Interface Set was discussed. The I/O pairs identified the execution paths through a routine containing various Ada constructs. The Ada constructs which were examined and implemented included assignment statements, If-Then-Else statements, loops, case statements, and procedure calls. The detailed design for IOGEN, the automated system for generating the I/O pairs, was presented. Each of IOGEN's four major components were discussed.

Finally, implementation enhancement details were discussed in order to ease modification efforts should the system be upgraded to support additional Ada constructs.

This thesis provides the initial design for generating I/O pairs for the CAIS. By no means is IOGEN a complete system. IOGEN makes an attempt in reducing the effort necessary to design the necessary test cases to validate routines in the CAIS. Not only can IOGEN be applied to CAIS routines but it can also be applied to Ada programs in general.

References

1. Ada Joint Program Office. Military Standard Common APSE Interface Set (CAIS). Department of Defense. 1985.

2. Booch, G. *Software Engineering with Ada.* Benjamin/Cummings Publishing Company, Menlo Park, CA, 1983.

3. Coleman, K.A. A CAIS Validation Methodology. MS Thesis, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1985.

4. Druffel, L. *Strategy for a DOD Software Initiative.* Defense Technology Center, Defense Logistics Agency, Alexandria, VA, 1982.

5. Facemire, J.L. and Lindquist, T.E. Using an Ada-based Abstract Machine Description of CAIS to Generate Validation Tests. *Washington Ada Symposium*, Washington D.C., 1985.

6. Hantler, S.L. and King, J.C. An Introduction to Proving the Correctness of Programs. ACM Computing Surveys, 8, 3, (1976), 331-353.

7. Kramer, J.F. et al. The CAIS Reader's Guide. IDA Memorandum Report M-150, 1985.

8. McGettrick, A.D. *Program Verification using Ada.* Cambridge University Press, New York, 1982.

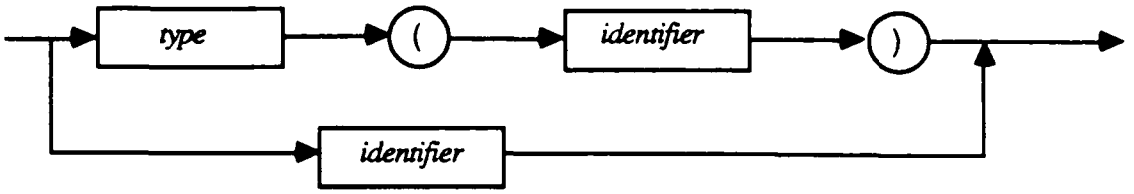
9. Myers, G.J. *The Art of Software Testing.* John Wiley and Sons, New York, 1979.

10. Pyster, A.B. *Compiler Design and Construction.* Van Nostrand Reinhold Company, New York, 1980.

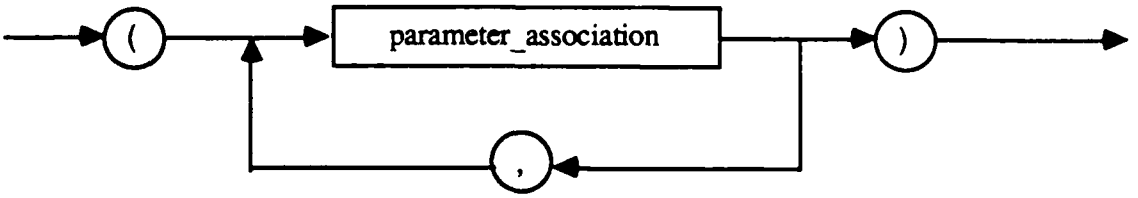
Appendix A

Syntax Diagrams

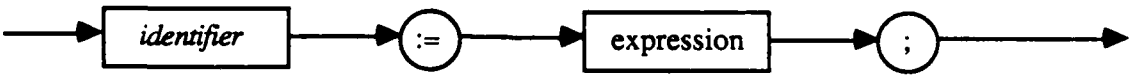
ACTUAL_PARAMETER



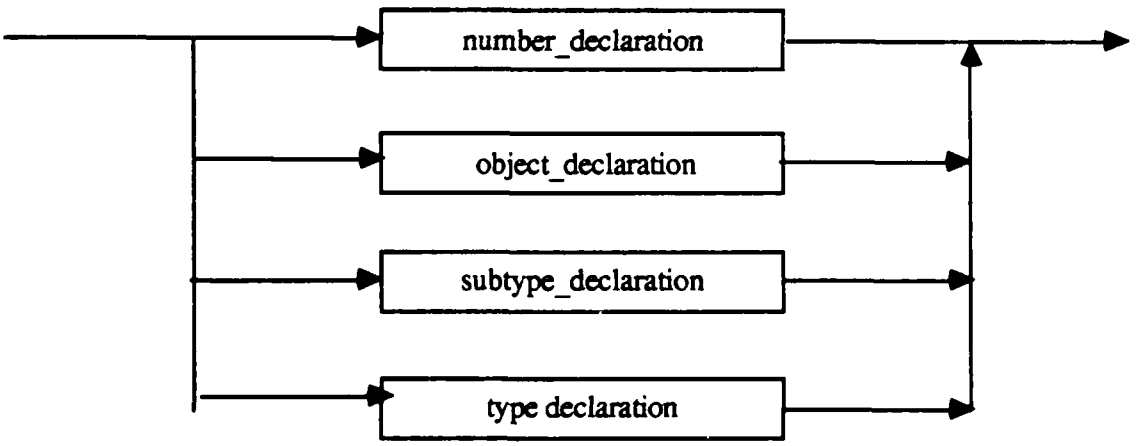
ACTUAL_PARAMETER_PART



ASSIGNMENT_STATEMENT

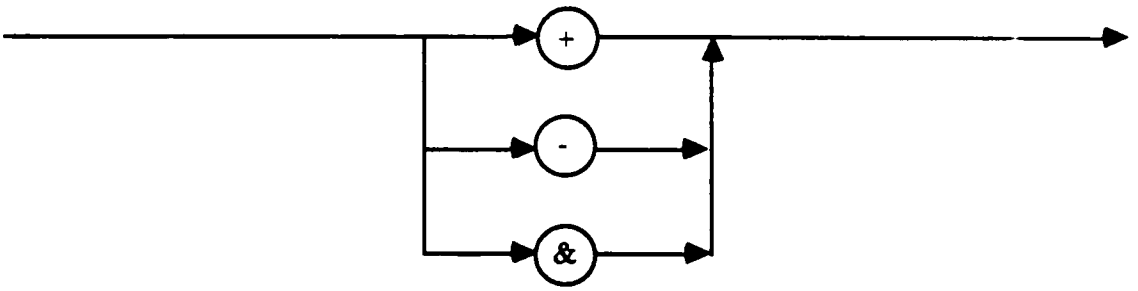


BASIC_DECLARATION

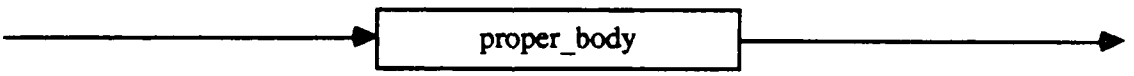


BASIC_DECL_ITEM

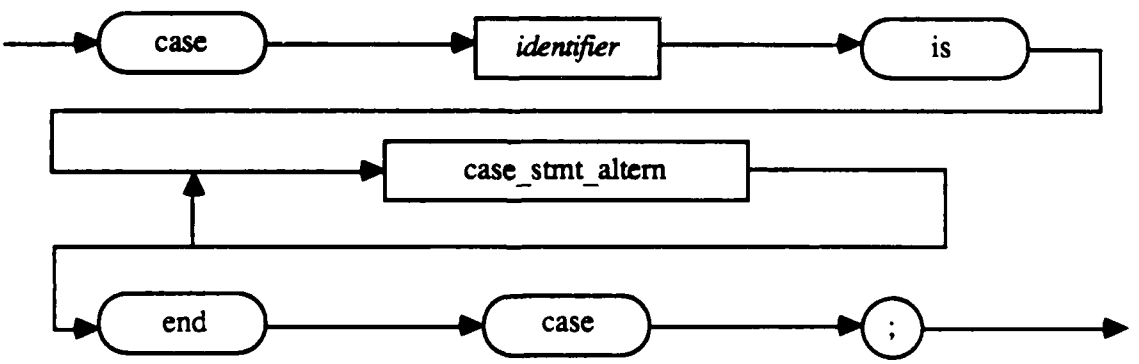
BINARY_ADDING_OP



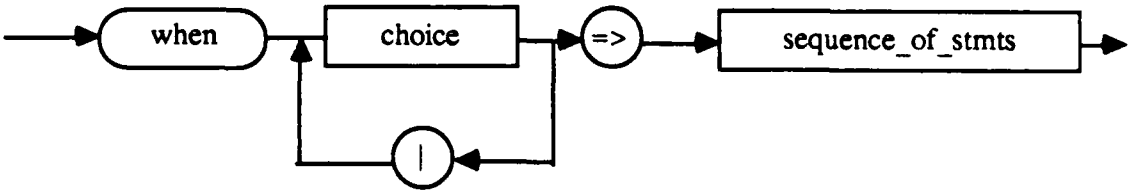
BODY



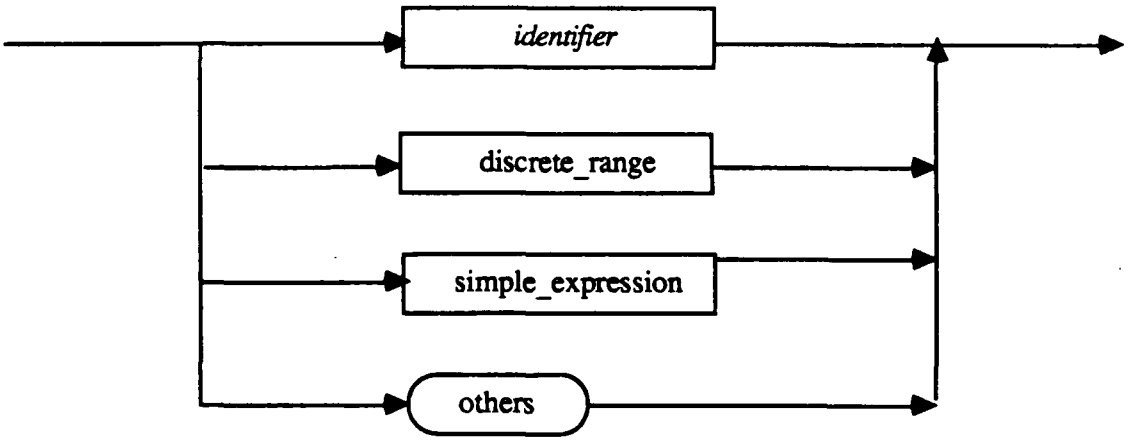
CASE_STATEMENT



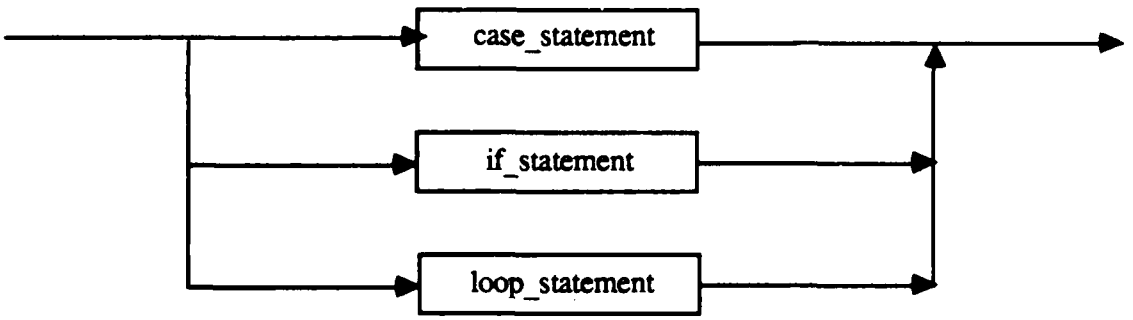
CASE_STMT_ALTERN



CHOICE

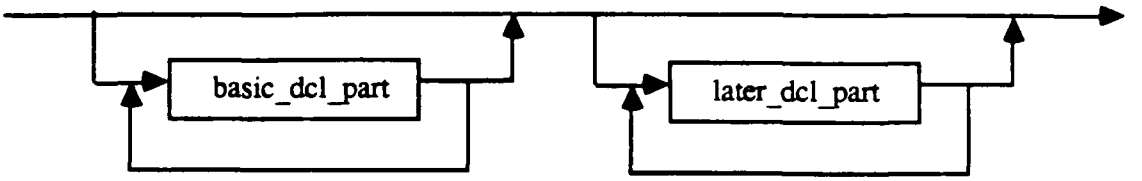


COMPOUND_STATEMENT

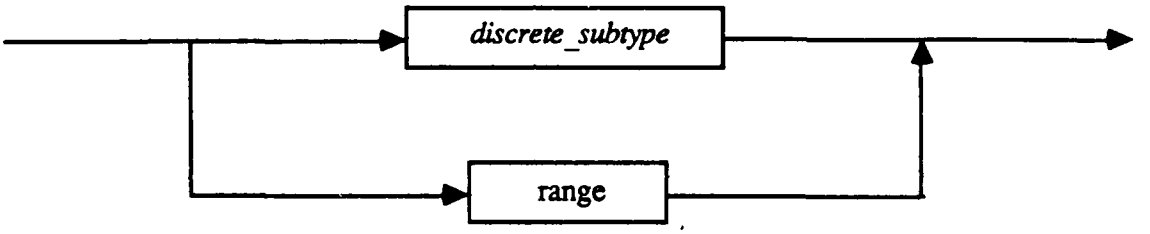


CONDITION

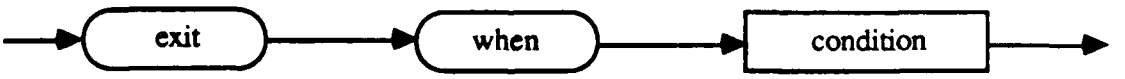
DECLARATIVE_PART



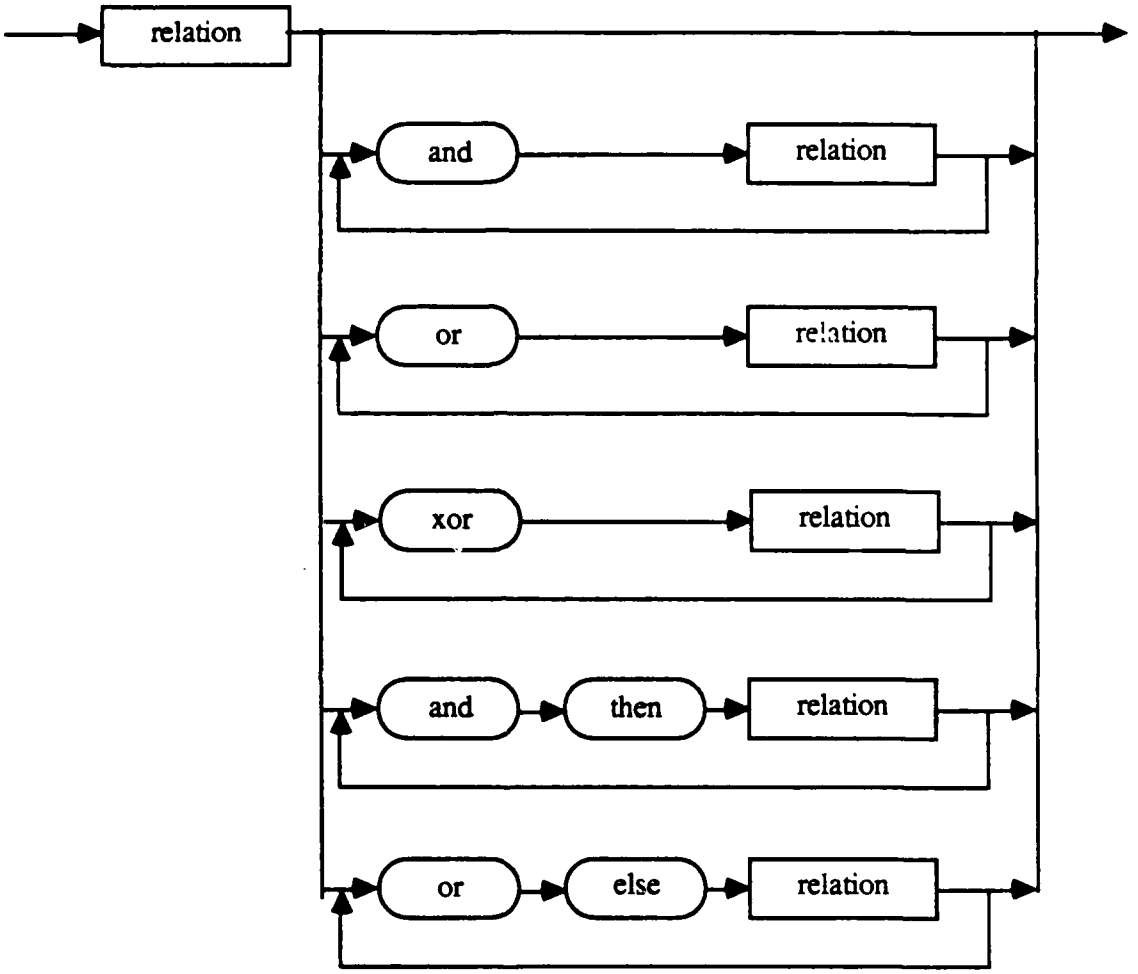
DISCRETE_RANGE



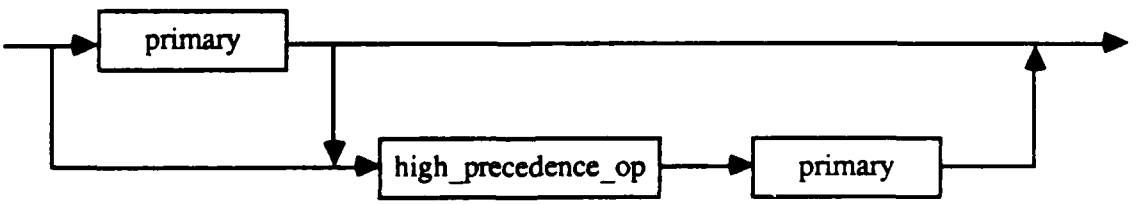
EXIT_STATEMENT



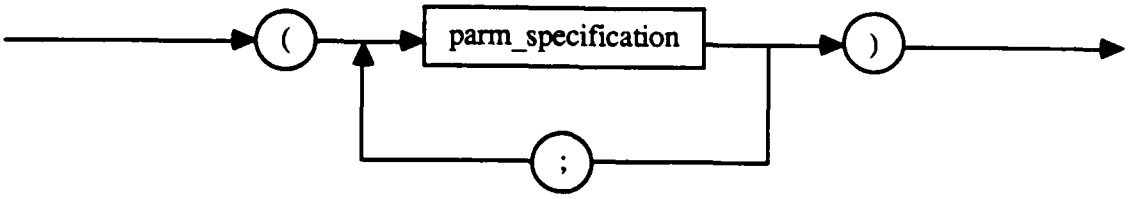
EXPRESSION



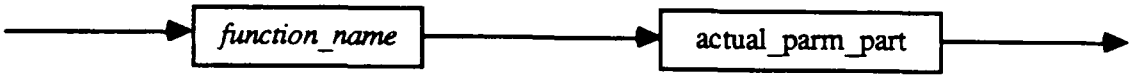
FACTOR



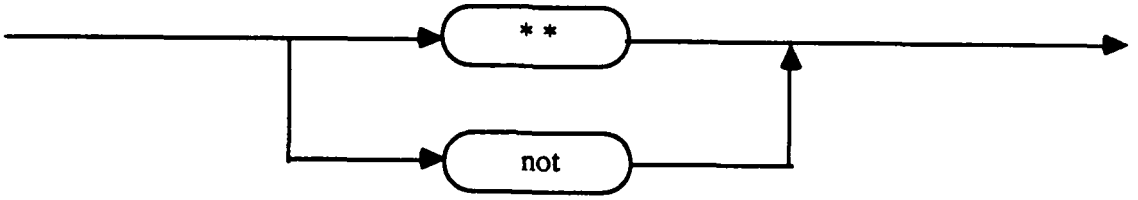
FORMAL_PART



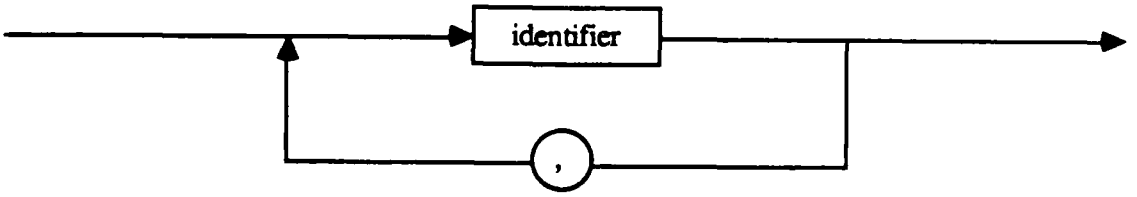
FUNCTION_CALL



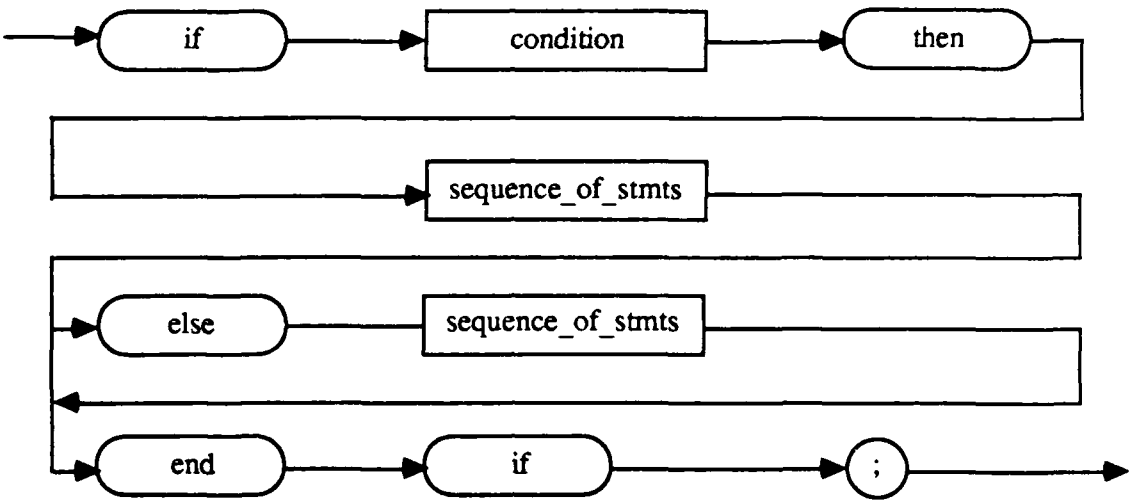
HIGH_PRECEDENCE_OP



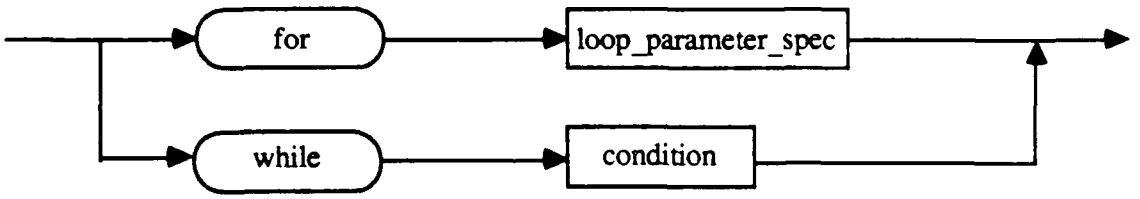
IDENTIFIER_LIST



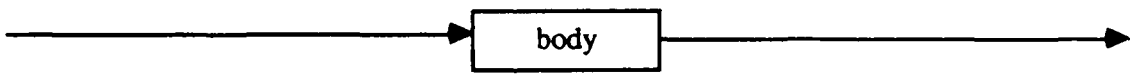
IF_STATEMENT



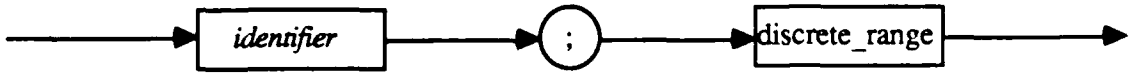
ITERATION_SCHEME



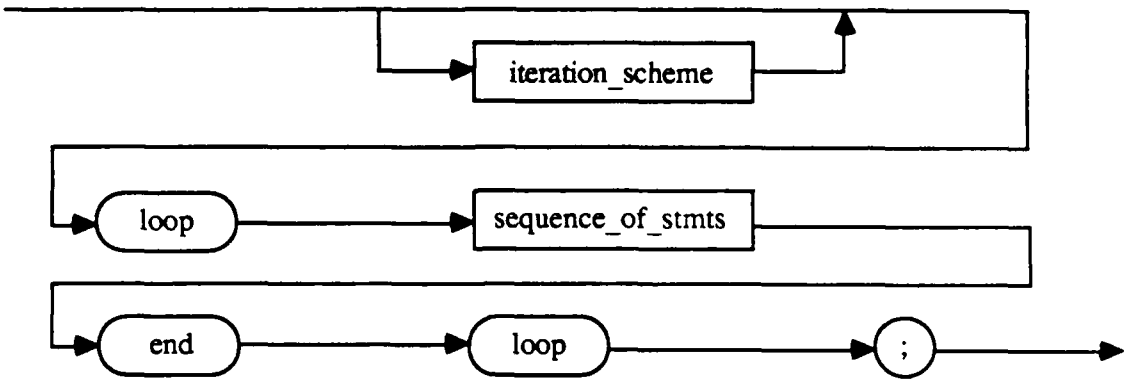
LATER_DECL_ITEM



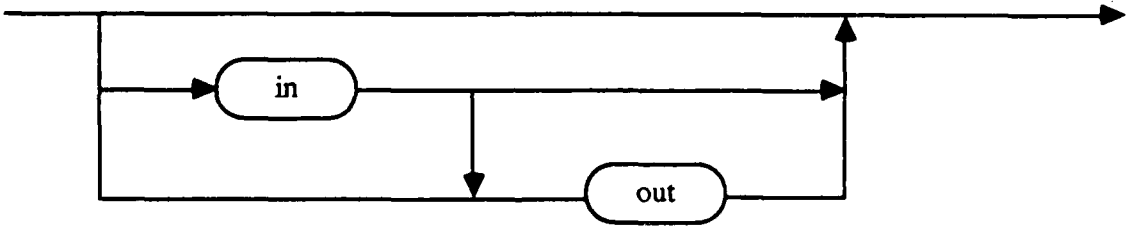
LOOP_PARAMETER_SPEC



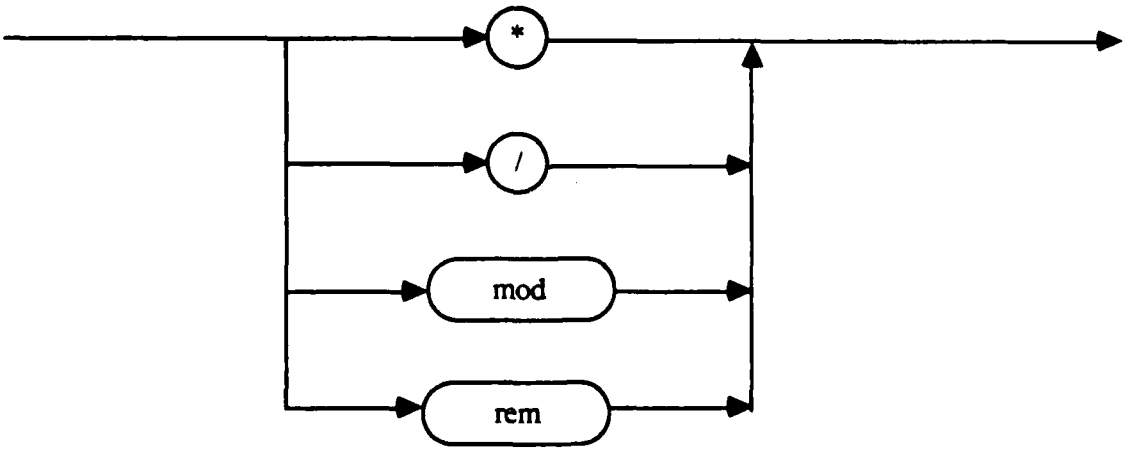
LOOP_STATEMENT



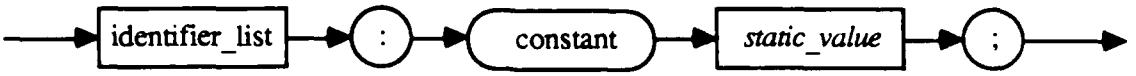
MODE



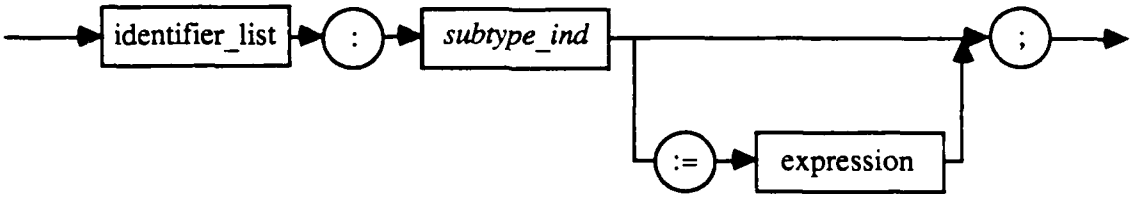
MULTIPLYING_OPERATOR



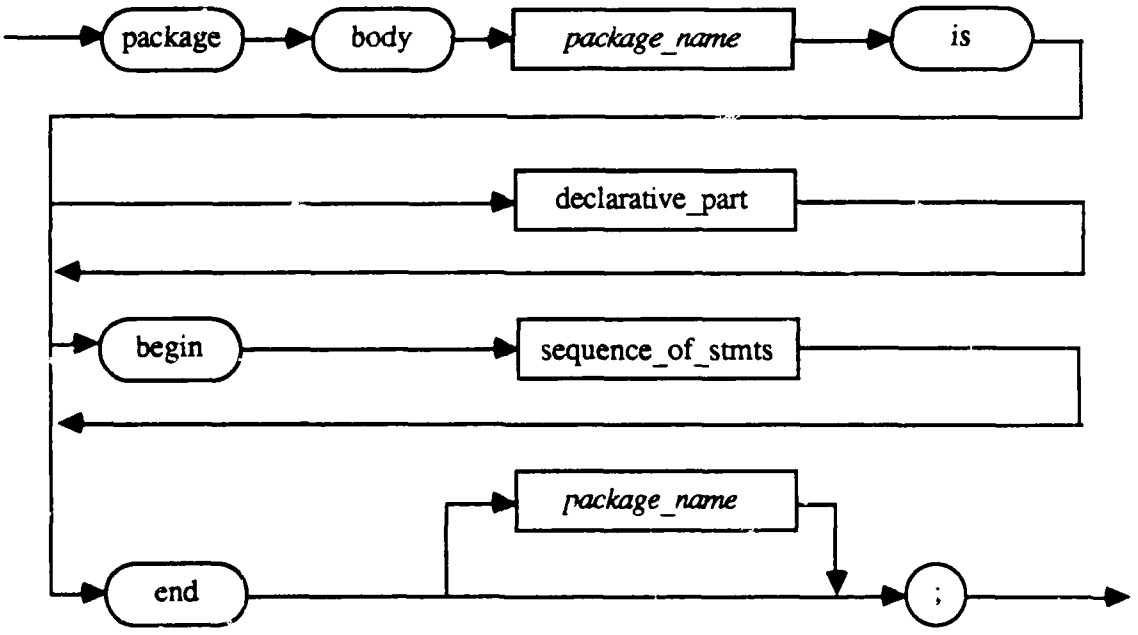
NUMBER_DECLARATION



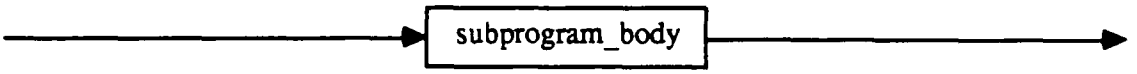
OBJECT_DECLARATION



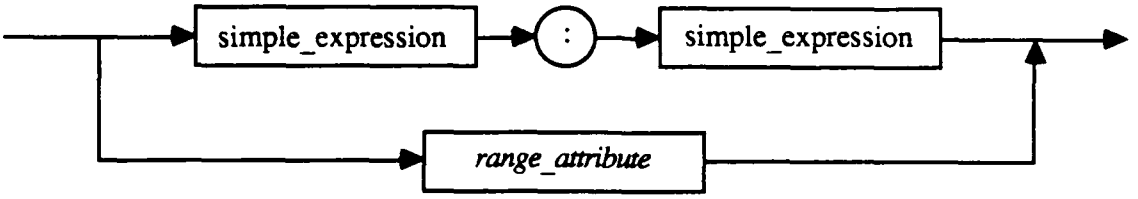
PACKAGE_BODY



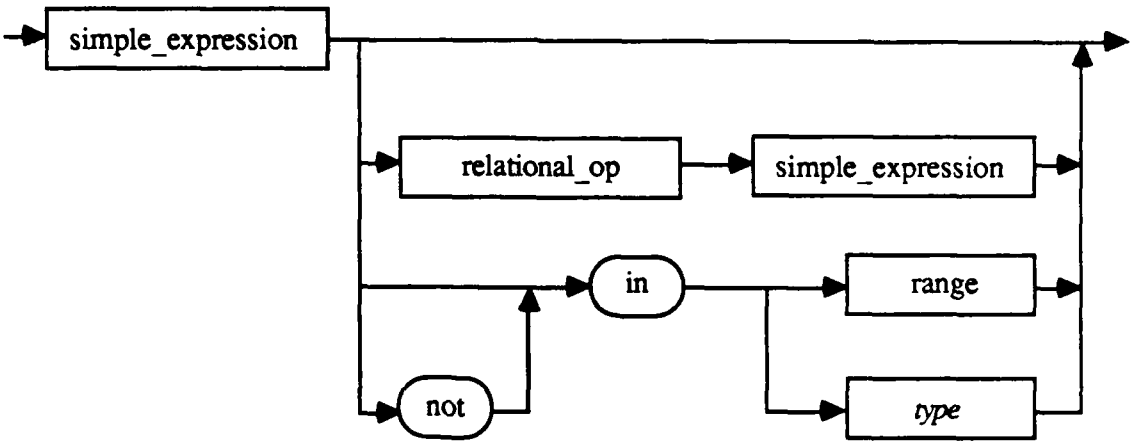
PROPER_BODY



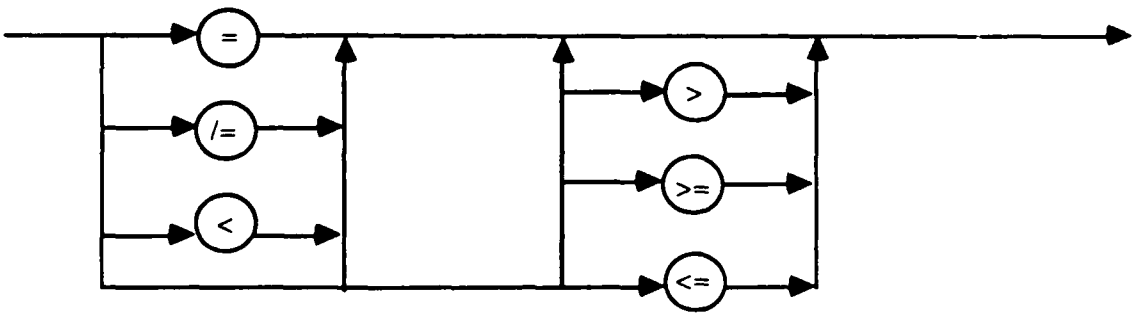
RANGE



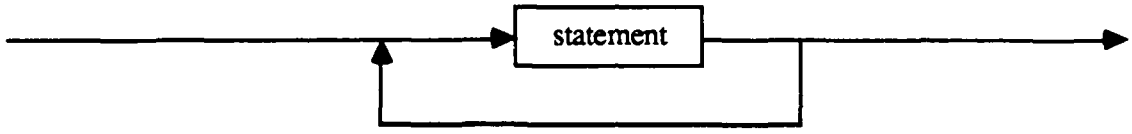
RELATION



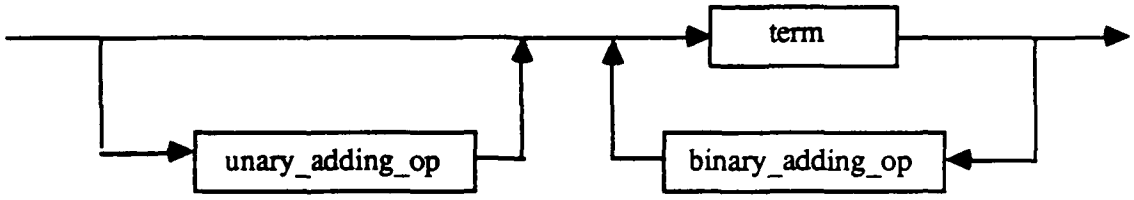
RELATIONAL_OP



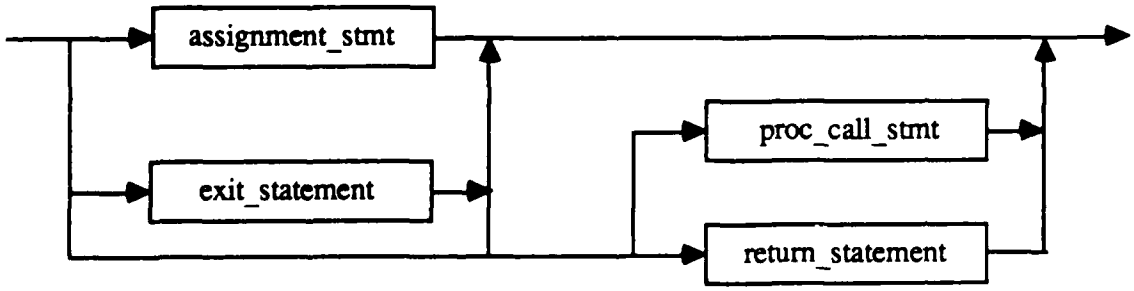
SEQUENCE_OF_STMTS



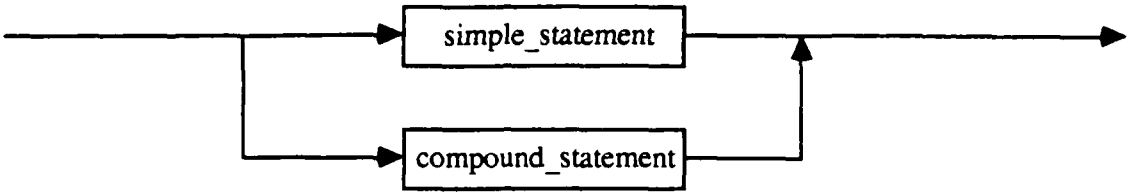
SIMPLE_EXPRESSION



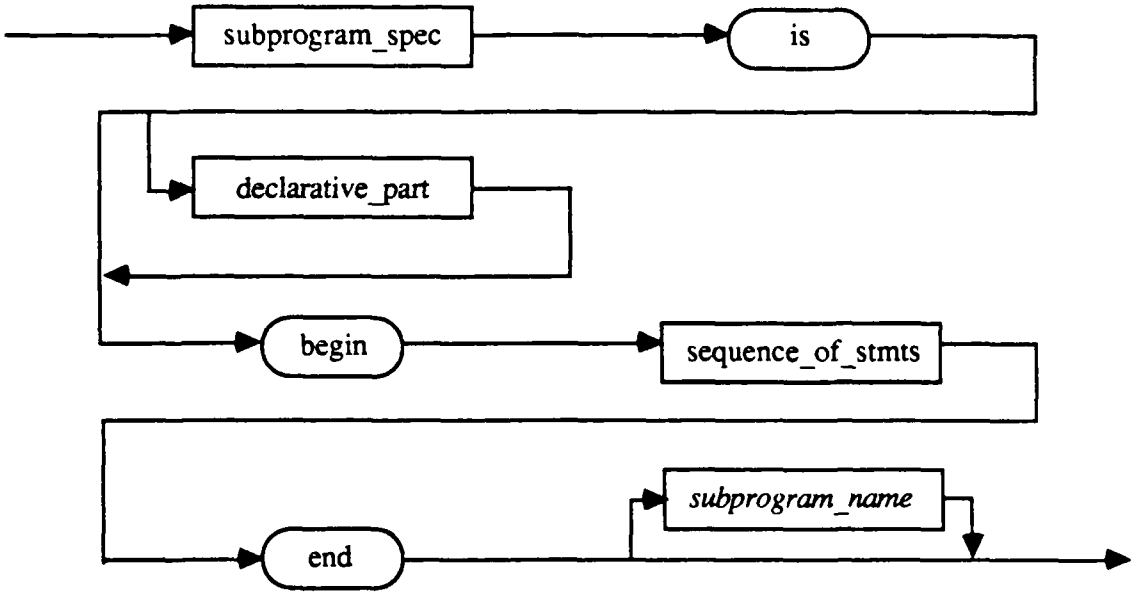
SIMPLE_EXPRESSION



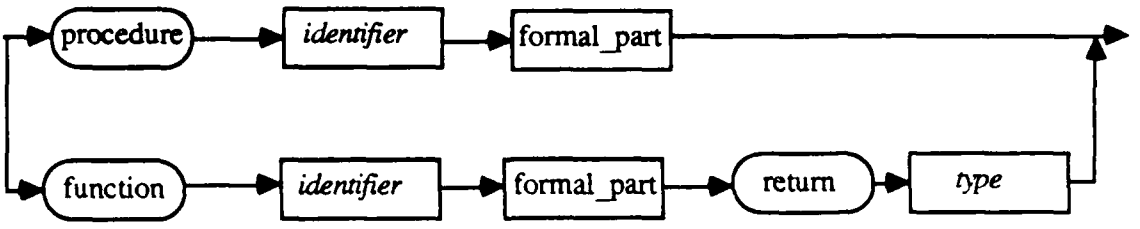
STATEMENT

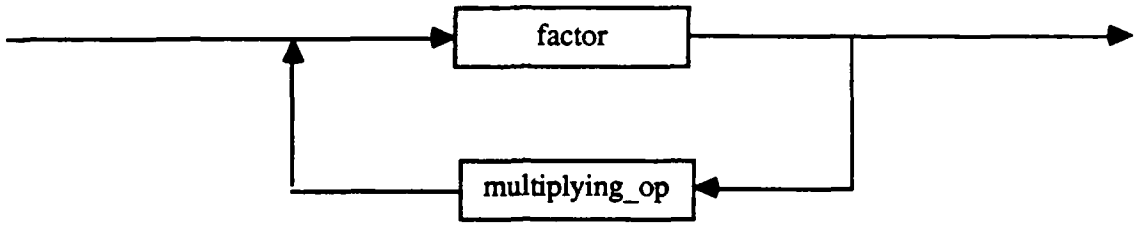


SUBPROGRAM_BODY

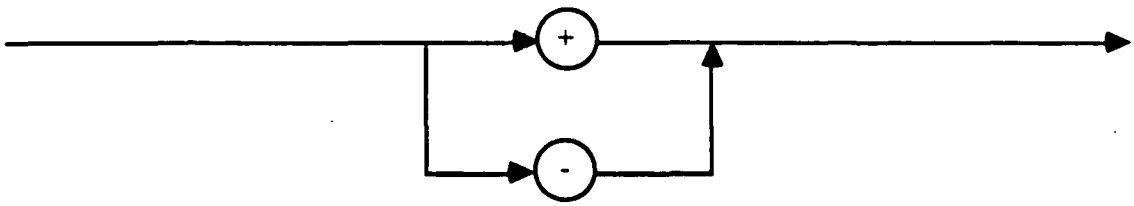


SUBPROGRAM_SPEC





UNARY_ADDING_OPERATOR



Appendix B

Sample Routine

```
[1]  package body CONVERT is

[2]  type DIGIT is -1 .. 9;
[3]  type DIGIT_CHAR is '0' .. '9';
[4]  type DIGIT_STRING is array <> of DIGIT_CHAR;

[5]  ACCUM  : integer := 0;
[6]  CHAR   : DIGIT_CHAR;
[7]  I      : integer := 1;
[8]  SIZE   : integer;
[9]  STR    : DIGIT_STRING;
[10] TEMP   : integer := 0;
[11] VALUE  : integer := 0;

[12] function ATOI ( C : in character ) return DIGIT is
[13] TEMP : DIGIT := 0;

[14] begin
[15]    if ( C in DIGIT_CHAR) then
[16]       TEMP := character'pos(C) - character'pos('0');
[17]    else
[18]       TEMP := -1;
[19]    end if;
[20]    return TEMP;
[21] end ATOI;

[22] begin
[23]    if I <= SIZE
[24]    then
[25]       CHAR := STR(I);
[26]       TEMP := ATOI(CHAR);
[27]       if TEMP /= -1 then
[28]          ACCUM  := ACCUM * 10 + TEMP;
[29]          I := I + 1;
[30]       end if;
[31]    end if;

[32] -- the integer value for the string of digits
[33] -- is contained in
[34]    VALUE := ACCUM;

[35] end CONVERT;
```

END

DTIC

9-86